STATISTICAL TESTS FOR CONNECTION ALGORITHMS FOR NEURAL NETWORKS

Daniel Hjertholm

NORWEGIAN UNIVERSITY OF LIFE SCIENCES
DEPARTMENT OF MATHEMATICAL SCIENCES AND TECHNOLOGY
MASTER THESIS 30 CREDITS 2013

Preface

This master thesis, written during the spring of 2013 at the Department of Mathematical Sciences and Technology, Norwegian University of Life Sciences, marks the end of my five year study program, Environmental Physics and Renewable Energy. The aim of this thesis has been to develop statistical test for some of the main connection algorithms used in neural networks.

The subject of this thesis is located somewhere in the intersection between the fields of statistics, computer science and neuroscience. I have some basic training in the first two, but the last one was completely new to me before I started this work. That was partly why I chose this subject. It was an opportunity to gain some insight into this exciting field, that is unraveling the mysteries of the human mind, and might come to shape technology in the future. Working on this project has been just as educational and interesting as I had hoped for.

I would like to thank my supervisor, Hans Ekkehard Plesser, and my co-supervisor, Birgit Kriener, for their guidance along the way, and for always taking the time to answer my questions thoroughly. Our fruitful conversations have been of great help. I would also like to thank my parents for their unconditional love and support throughout my life, and for cheering for me from the sidelines through this project. Last, but not least, I would like to thank Elina. You have been very supportive, and always patient, even through the most stressful times, when I was glued to my computer for hours on end, unable to tear myself away.

Abstract

Simulations of increasingly sophisticated neural network models have accelerated the progress of neuroscience, but have also led to an increased reliance on the simulation software. Testing the quality of this software therefore becomes important. Here we develop test strategies for some of the most common probabilistic connection algorithms used in simulators, and implement these as Python based test suites. We develop approaches to alleviate the problems of statistical software testing, i.e., the unavoidable occurrence of false positives and negatives. The tests are developed for the NEST simulator, but can easily be adapted to work with analogous connection algorithms in other simulators.

For random connections with predefined in- or out-degree, observed random degrees are compared with expectation using Pearson's chi-squared test. For networks with structure in two- or three-dimensional space, i.e., with a distance-dependent connection probability, the Kolmogorov-Smirnov (KS) test is used to compare the empirical distribution function (EDF) of distances between connected source-target pairs with the expected cumulative distribution (CDF), obtained by numerical integration of the normalized product of the radial distribution of nodes and the distance-dependent connection probability (kernel). A Z-test comparing the total number of connections with the expectation is also implemented. For all three of these tests, a two-level test can be used, comparing the distribution of p-values from multiple tests of individual network realizations with the expected uniform distribution, using the KS test. This approach results in greatly increased sensitivity. For automated tests used in test suites, an adaptive approach is proposed. Here, one test is performed, and if the result is suspicious, the more thorough two-level test is performed. This approach is fast, as the two-level test is only invoked for a fraction of the test cases under normal circumstances. It also results in a very low rate of false positives.

The probabilistic connection algorithms of NEST were tested under a variety of conditions, e.g. with different network sizes, different number of virtual processes (VPs) and different distance-dependent connection probability functions (kernels). No evidence of any error or bias was found in the algorithms. The test strategies themselves were shown to detect an array of small errors and biases when these were deliberately introduced into the algorithm.

Sammendrag

Simuleringer av stadig mer sofistikerte nevrale nettverksmodeller har vært med på å drive nevrovitenskapen fremover, men har samtidig ført til en økt avhengighet av simuleringsprogramvare. Å teste kvaliteten til denne programvaren der derfor viktig. Her utvikler vi strategier for testing av noen av de mest brukte koblingsalgoritmene, og vi implementerer disse som Python-baserte testpakker. Vi utvikler metoder for a redusere hyppigheten av type I og type II feil. Testene er utviklet for simulatoren NEST, men kan modifiseres for å fungere med tilsvarende koblingsalgoritmer i andre simulatorer.

For tilfeldige koblinger med forhåndsbestemt inn- eller utgrad sammenlignes de observerte tilfeldige gradene med forventningsverdier ved hjelp av Pearsons kjikvadrattest. For nettverk med romlig struktur, dvs. nettverk med en koblingssannsynlighet som avhenger av avstand, brukes Kolmogorov-Smirnov-testen (KS-testen) til å sammenligne den empiriske kumulative fordelingen av avstander mellom sammenkoblede node-par med den forventede kumulative fordelingen, funnet ved numerisk integrasjon av det normaliserte produktet av den radiale fordelingsfunksjonen av noder og den avstandsavhengige koblingssannsynligheten. En Z-test som sammenligner det totale antallet koblinger med forventningsverdien er også implementert. For alle disse testene kan en to-nivå-test anvendes. Denne sammenligner fordelingen av p-verdier fra flere tester av individuelle nettverksrealiseringer med den forventede uniforme fordelingen ved hjelp av KS-testen. Dette gir en betraktelig økt sensitivitet. For automatiserte tester brukt i testpakker foreslår vi en adaptiv løsning. Denne går ut på at en enkel test kjøres, og kun dersom resultatet er mistenkelig kjøres den mer grundige to-nivå-testen. På denne måten kan automatiserte tester kjøre fort, siden to-nivå-testen normalt kun vil kjøres for en liten andel av test-tilfellene. I tillegg får vi få feilaktige godkjennelser (type I feil).

De tilfeldige koblingsalgoritmene i NEST ble testet under ulike forhold, for eksempel med forskjellig nettverkstørrelse, forskjellig antall virtuelle prosesser og foreskjellige avstandsavhengige koblingssannsynligheter. Det ble ikke funnet noe bevis på feil eller skjevheter i algoritmene. Det ble vist at teststrategiene oppdaget en rekke feil og skjevheter når disse bevisst ble lagt inn i algoritmene.

Contents

Preface **i**

Abstract **iii**

Sammendrag **v**

List of Figures **x**

List of Tables **xi**

List of Symbols **xiii**

1 Introduction **1**

2 Statistical testing **7**
2.1 Pearson's chi-squared test 8
2.2 The Kolmogorov-Smirnov test 11
2.3 The Z-test 13

3 Distribution of connections **15**
3.1 Random convergent connections 15
3.1.1 Implementation 16
3.1.2 Results 17
3.2 Random divergent connections 22
3.2.1 Implementation 22
3.2.2 Results 23
3.3 Automated test procedure 23
3.3.1 Implementation 24

4 Spatially structured networks **27**
4.1 Two-dimensional space 27
4.1.1 Implementation 30
4.1.2 Results 32
4.2 Three-dimensional space 35

4.2.1 Implementation 38
4.2.2 Results 38
4.3 Automated test procedure 40
4.3.1 Implementation 41

5 Discussion 43

A NEST - A short tutorial 47

B Test script for random convergent connections 59

C Test script for random divergent connections 63

D Automated test suite for random convergent and divergent connections 67

E Test script for 2D spatially structured network 69

F Test script for 3D spatially structured network 73

G Automated test suite for spatially structured networks 77

References 81

List of Figures

1.1 A random divergent connection pattern 4
1.2 A spatially structured network 5

2.1 CDF for $X \sim \mathcal{U}(1,0)$ and EDF of pseudorandom numbers from a PRNG 12

3.1 Histogram and EDF of p-values from 10,000 chi-squared GOF tests 18
3.2 EDF of 100 p-values from the two-level tests procedure 19
3.3 The effect of small parameter values on the EDF of p-values . . 20
3.4 Distribution of p-values after introducing a bias into the connection algorithm 21

4.1 Illustration of the derivation of the expression for the radial distribution function 28
4.2 The effect of the mask on the radial distribution function 29
4.3 Exemplary PDF, CDF and connectivity pattern for a network in 2D space, using a Gaussian kernel 31
4.4 Theoretical and empirical PDF and CDF of source-target distances using a constant kernel 32
4.5 Connectivity pattern with periodic boundary conditions and a source node shifted away from the layer's center 33
4.6 Detection rate as a function of network size 34
4.7 Illustration of the effect of a cubic mask on the radial distribution function 36
4.8 Illustration of the derivation of the surface area E of the intersection between adjacent spherical caps 37
4.9 Exemplary PDF, CDF and connectivity pattern for a network in 3D space, using a Gaussian kernel 38
4.10 Theoretical and empirical PDF and CDF of source-target distances using a constant kernel 39

A.1 Example of free two-dimensional layer 53
A.2 Example of free three-dimensional layer 54

A.3 Example of connectivity pattern 56
A.4 Distribution of distances between source node and connected target nodes 57

List of Tables

3.1 Default parameter set for reported results 17
3.2 Default parameter set for sensitivity testing 21

A.1 Neuron models in NEST 48
A.2 Devices in NEST 49
A.3 Synapse models in NEST 50
A.4 Kernels in NEST Topology 55

List of Symbols

α	level of significance / probability of making a type I error	p. 7
β	probability of making a type II error	p. 7
$\mathcal{U}(a,b)$	uniform distribution on the interval $[a,b]$	p. 7
A_i	outcome of a trial with a fixed number r of possible outcomes	p. 9
n	number of trials	p. 9
r	number of possible outcomes A_i	p. 9
p_i	probability of outcome A_i	p. 9
$\boldsymbol{p}$	vector of outcome probabilities p_i	p. 9
ν_i	frequency of outcome A_i	p. 9
$\boldsymbol{\nu}$	vector of frequencies	p. 9
$\mathrm{E}(\nu_i)$	expected frequency of outcome A_i	p. 9
$\mathrm{E}(\boldsymbol{\nu})$	vector of expected frequencies	p. 9
X^2	Pearson's test statistic	p. 10
χ^2	the chi squared distribution	p. 10
$F(x)$	cumulative distribution function of a random variable	p. 11
$S_n(x)$	empirical distribution function that converges to $F(x)$	p. 11
$F_0(x)$	hypothesized cumulative distribution function	p. 11
D_n	Kolmogorov-Smirnov test statistic	p. 11
Z	standard score (z-score)	p. 13
$\mathcal{N}(\mu,\sigma^2)$	normal distribution with mean μ and variance σ^2	p. 13
C	fixed in or out degree (number of connections) in network with random convergent or divergent connections, respectively	p. 15
N_s	number of source nodes in network	p. 15
N_t	number of target nodes in network	p. 15
n_runs	number of times to run chi-squared test in two-level test	p. 17
L	side length of square or cubic layer	p. 27
N	number of nodes in spatially structured network	p. 27

D	distance from center node	p. 27
D_{ij}	distance from node i to node j	p. 27
ρ_0	areal or volumetric node density for networks in 2D or 3D space, respectively	p. 28
$\rho(D)$	radial distribution function (RDF)	p. 28
$\mathcal{P}(D)$	distance-dependent connection probability (kernel)	p. 29
$f(D)$	probability density function (PDF) of source-target distances	p. 29
$F(D)$	cumulative distribution function (CDF) of source-target distances	p. 29
$H(x)$	the Heaviside step function	p. 30
C	total number of connections in spatially structured network	p. 30

Chapter 1

Introduction

The brain is a truly remarkable machine. It routinely performs tasks that are impossible for even the most powerful supercomputer, and it does so in a highly energy-efficient manner. It can analyze patterns extremely efficiently, interpret them, and produce appropriate behavioral responses. It can store enormous amounts of information, allowing us to recollect things like phone numbers, familiar faces, songs and episodes from early life. But perhaps even more impressively, the brain is capable of producing consciousness, and allows us to experience our own thoughts and feelings.

Scientists' desire to unlock the mysteries of the brain is obvious. Not only would it allow us to better understand, and perhaps cure, the hundreds of different brain diseases that exist, such as depression, Alzheimer's disease, Parkinson's disease, epilepsy, and migraine. The knowledge of how the brain solves complex problems would likely also result in radical changes to how computers work. New storage technologies that mimic the way the brain stores and recollects memories might emerge, and computers might finally be able to learn the way humans do. Artificial intelligence (AI) might seem a lot less artificial.

However, understanding the brain is one of the greatest challenges facing scientists today. A great deal is understood about how single neurons function, how they communicate with other neurons through synapses, and how these synapses are formed and change over time. What is not well understood is how brain function emerges from billions of neurons communicating with each other over trillions of synapses. One way to address this problem is to simulate the activity of a large number of neurons in a computer. With such a simulation, the effects of small changes in neurons and synapses on the dynamics of large networks can be studied. The Human Brain Project (HBP), recently awarded one billion euros by the European Commission (Abbott 2013), aims to simulate a full scale human brain within a decade (The Human Brain Project Preparatory Study Consortium 2012). With this flagship project engaging and inspiring the scientific community, the importance of simulations in brain

science is likely to continue to increase in the years to come.

A number of neural network simulators are available, such as NEURON, GENESIS, Brian, and NEST (Brette et al. 2007). This thesis will focus on NEST (Gewaltig and Diesmann 2007), a popular simulation environment for simulating large networks of point neurons. As scientists are increasingly relying on tools like these to make new discoveries, the need for quality checking of the software increases. There is always a chance that a mistake has crept into a piece of computer code, one that does not make the program crash, but causes erroneous scientific results. In a famous example, a paper detailing the structure of a protein called MsbA (Chang and Roth 2001), had to be retracted because the reported structure turned out to be wrong. The reason for the incorrect result was that two columns of data were flipped in the computer program that derived the protein structure (Miller 2006). It took more than five years before the mistake was detected, and by that time the paper had been cited by 364 publications, according to Google Scholar. Four other papers also had to be retracted due to the same malfunctioning computer program.

A good way to detect these kinds of mistakes in computer code is through unit testing. The idea here is to divide the program into small units that can be tested rigorously by comparing the units output with our expectations (Huizinga and Kolawa 2007). While this approach has its merits, tests will not be able to detect all conceivable error that could occur. In the words of Dijkstra et al. (1970), "Program testing can be used to show the presence of bugs, but never to show their absence!".

When testing probabilistic algorithms using statistical tests, some additional challenges arise. For instance, the tests will occasionally give false positives, i.e., they will report a problem with the tested algorithm where none exists. One can not get completely rid of false positives, but strategies can be devised to reduce the frequency with which they occur. Statistical tests will also give false negatives, i.e., they will fail to report true problems. Again, this is unavoidable, all we can do is to try to increase the sensitivity of the tests, i.e., their ability to detect problems. This ties in with a more general problem; that of testing randomness.

To generate a sequence of seemingly random numbers, computer software relies on pseudorandom number generators PRNGs (L'Ecuyer 2004). The sequences they produce is not truly random, but generated by a deterministic algorithm (hence the prefix "pseudo"). For most applications, however, the numbers are sufficiently unpredictable as to be considered random, depending on the specific algorithm used and how it is implemented. There are several ways to test the output of a PRNG. One approach is to look for a systematic bias by comparing the expected distribution of numbers with an observed sample distribution using some goodness-of-fit (GOF) test, such as the Kolmogorov-Smirnov (KS) test or the Anderson-Darling (AD) test, but this approach will not detect other kinds of patterns such as clustering of the

produced numbers, as long as the clusters are distributed relatively evenly. Other methods may pick up such clustering, but yet other unforeseeable patterns might still go undetected. As argued by L'Ecuyer and Simard (2007), no statistical test or battery of tests can guarantee that the output of a PRNG will be sufficiently random under all circumstances. Some application might cause some structure to emerge due to an artifact of the PRNG's algorithm.

When testing probabilistic algorithms using statistical tests, we are faced with the same problem. No statistical test or battery of tests can prove beyond any doubt that the algorithm is without biases or patterns that affects the output of a computer program. Still, the more tests the algorithms can pass, the greater our confidence in them, and hence the scientific findings based on them.

Connectivity patterns in neuronal network models

A neural network can be described as a directed, weighted graph where the nodes are neurons, or sometimes devices, and the edges are connections or synapses between them. Events can be transmitted in one direction over the connections. These events are action potentials (spikes) or other types of signals that can be transmitted over synapses.

In any type of network modeling, the network structure must be specified. In principle, this could be done by listing all the nodes and their connections. This kind of specification, however, is not very manageable for us humans. To be able to work with network models, discuss them, and share them, higher-level descriptions of connection patterns are needed. Populations of nodes can then be connected following some basic rules to create these patterns. Ways to unambiguously describe and document connection patterns have been lacking, but recent efforts have been made to standardize their terminology and notation (Nordlie and Plesser 2010; Djurfeldt 2012; Crook et al. 2012). Since NEST is used in this thesis, we will mainly use NESTs terminology. We will now describe the relevant connection patterns and related concepts and terminology.

A **random divergent connection** between a source population and a target population is defined as the connection pattern resulting from connecting each node in the source population to a prescribed number C of randomly drawn target nodes. An example of this kind of network is shown in Figure 1.1. The target nodes are drawn with replacement, meaning multiple connections can exists between any pair of nodes (multapses). If the source and target populations are the same population, nodes will also be able to connect to themselves (autapses). It is possible to disallow multapses and autapses in NEST, but we will assume they are allowed.

In a **random *convergent* connection**, C *source nodes* are randomly drawn, with equal probability, for each target node. The in-degree is now C

for all target nodes, while the source nodes might have different out-degrees.

Network models can also have spatial structure. In such models, connection probabilities, as well as connection properties such as weight and delay, can be a function of the distance between the source and the target node. In this way, connectivity patterns that mimic observed axonal projection patterns can be created. Thus, when referring to a **spatially structured network** in this thesis, we mean a network whose connection probabilities depend on source-target distance. In spatially structured networks, populations of nodes can exist in two- or three-dimensional space. The distance-dependent connection probability function is referred to as the **kernel**. Only nodes inside an area or volume called the **mask** are eligible connection targets. The location of the mask is usually given relative to the node considered. A two-dimensional spatially structured network is shown in Figure 1.2, where a square mask and a Gaussian kernel are used.

Aims and organization of this thesis

The aim of this thesis is to develop statistical tests for the probabilistic connection algorithms used in neural network simulators, both for networks with and without spatial structure. Further, an automated test suite that can run relatively quickly, with a low rate of false positives, possibly as part of the automated testing in a continuous integration system, is developed. To demonstrate the utility of the test procedures, they are used to test the connection algorithms of NEST. It is worth emphasizing that the test suites can easily be modified to work with data obtained from any other simulator that shares the basic connection schemes.

Chapter 2 will start with a short introduction to the statistical tests that will be used later, and introduce the concept of a two-level test procedure.

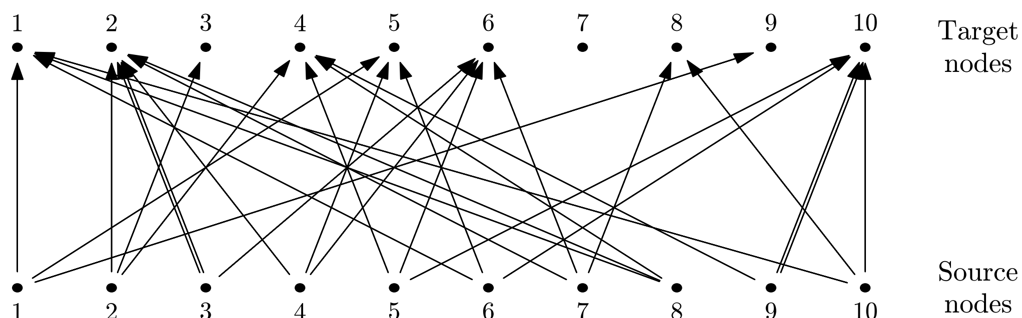


Figure 1.1: Example of how a random divergent network might look. All source nodes have an out-degree of 3, while the target nodes have different in-degrees.

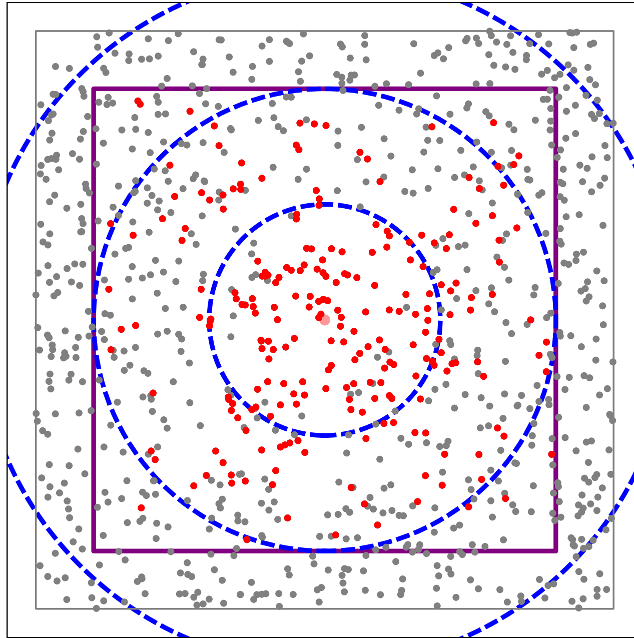


Figure 1.2: Example of how a spatially structured network might look. 1,000 nodes are scattered across the layer. The centered source node is connected to the red nodes, but not the grey. The mask is shown in purple, and the blue rings mark σ, 2σ and 3σ from the Gaussian kernel.

In Chapter 3, a test procedure for random connections with predefined in- or out-degree is developed. NEST-specific implementations are discussed, and results are presented, both for convergent (Section 3.1) and divergent (Section 3.2) connections. In Section 3.3, an automated test procedure is proposed and implemented for NEST.

Networks with spatial structure are discussed in Chapter 4. Test strategies are developed and implemented for NEST, and results are presented, first for two-dimensional space in Section 4.1, then for three dimensions in Section 4.2. A general discussion about findings and perspectives for future research is found in Chapter 5.

As the tests proposed in this thesis are implemented for NEST, there will be references to functions in NEST. The reader might therefore want to take a look at the NEST tutorial found in Appendix A before proceeding.

Chapter 2

Statistical testing

The p-value approach to statistical hypothesis testing consists of the following five steps (Ewens and Grant 2004).

Step 1 is to state the *null hypothesis* (H_0) and the *alternative hypothesis* (H_1). The aim is to either reject or accept H_0. Accepting H_0 does not mean that it is necessarily true, only that there was insufficient evidence against it.

Step 2 is to determine the *level of significance* α. This is the probability of making a *type I error*, i.e., rejecting H_0 when it is true. We obviously want this probability to be small, but a too small α will increase the probability β of making a *type II error*, i.e., accepting H_0 when it is false. The choice of α will therefore depend on the situation and what type of error is most important for us to avoid.

Step 3 is to decide on a *test statistic* T. Which test statistic is used depends on the situation.

Step 4 is to compute the value t_{obs} of this test statistic from the observations.

Step 5 is to compute the p-value, i.e., the probability that the test statistic T would have a value at least as extreme as the observed value t_{obs} under H_0. If the alternative hypothesis corresponds to large values of T, the p-value is calculated as $\text{Prob}(T \geq t_{\text{obs}})$. Thus, roughly speaking, a small p-value indicates that H_0 is unlikely. The p-value is compared with the chosen level of significance α (typically 0.05 or 0.01). If the p-value it is smaller, H_0 is rejected, otherwise, H_0 is accepted.

When the test statistic is continuous, the p-value is a continuous, random variable with a uniform distribution $\mathcal{U}(0,1)$ under H_0 (Ewens and Grant 2004),

meaning it will satisfy the equation

$$\text{Prob}(p\text{-value} \leq x \mid H_0 \text{ true}) = x \tag{2.1}$$

for $x \in [0, 1]$. When the test statistic is *discrete*, the p-value is also discrete, but it is *not* discrete *uniform* under H_0. The reason is that a p-value that satisfies the equation above might not exist. Instead, the p-value will satisfy

$$\text{Prob}(p\text{-value} \leq x \mid H_0 \text{ true}) \leq x. \tag{2.2}$$

The deviation from the uniform distribution is large when data is sparse, but for larger data sets, the deviation is often negligible.

In certain situations, a *two-level test procedure* is advantageous. First, the expected distribution is compared with the empirical distribution observed, using some goodness-of-fit (GOF) tests. In accordance with classical hypothesis testing, a failure to pass such a test at some level of significance α would result in the rejection of the null hypothesis H_0 that the empirical frequencies follow the expected distribution, with a probability α of making a type I error. Successfully passing a test would typically cause H_0 to be accepted, with a probability β of making a type II error. β is often quite large. We might therefore want to run the test several times before we feel confident that H_0 can be accepted. When doing experiments on a computer, the cost of re-running the test is usually small. We can, therefore, run the experiment a large number of times, and check whether the test fails the expected fraction α of the tests. But this way we would lose a lot of information, and it is arguably not the best approach in this situation. In the case where the p-value is (sufficiently) uniform under H_0, we can instead test the observed p-values generated against the expected $\mathcal{U}(0, 1)$. This approach, sometimes referred to as a *two-level test procedure*, is similar to the one used by L'Ecuyer and Simard (2007) to test PRNGs. The test of the p-values will of course have its own p-value, which in turn could be compared to the expected uniform distribution using some GOF test, and so on, repeating endlessly. Nevertheless, it should suffice to do one test against uniformity, and compare the resulting p-value to some predefined level of significance α. The main advantage of this two-level approach is that if a connection algorithm passes the test, our confidence in it will be much greater than if only a single test was performed.

The GOF test used in the first step of the procedure will vary depending on the data. The tests we will use are introduced below. For the second step (the test of GOF of the p-values to $\mathcal{U}(0, 1)$) the Kolmogorov-Smirnov test is used.

2.1 Pearson's chi-squared test

The following section is based on *A Guide to Chi-Squared Testing* (Greenwood and Nikulin 1996). Pearson's chi-squared test, sometimes ambiguously referred

to as "the chi-squared test", is a test of the null hypothesis that there is a good fit between some theoretical distribution and the observed data.

Consider an experiment, or *trial*, resulting in one outcome, A_i, of r possible outcomes, $A_1, A_2, ..., A_r$. The probability of outcome A_i is p_i. We conduct n independent trials, indexed $k = 1, 2, ..., n$. We define a random variable

$$\mu_{ki} = \begin{cases} 1 & \text{if outcome } A_i \text{ occurred in } k\text{th trial} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

for $i = 1, 2, ..., r$. We now define a vector

$$\boldsymbol{\mu_k} = (\mu_{k1}, \mu_{k2}, ..., \mu_{kr}), \quad k = 1, 2, ..., n \tag{2.4}$$

which, for each trial k, describes which outcome occurred. It has only one nonzero component, the one indexed i, which is equal to 1. The *frequency* of outcome A_i is defined as

$$\nu_i = \sum_{k=1}^{n} \mu_{ki}. \tag{2.5}$$

All these frequencies must satisfy the condition $\nu_1 + \nu_2 + ... + \nu_r = n$. The *vector of frequencies* becomes

$$\boldsymbol{\nu} = \sum_{k=1}^{n} \boldsymbol{\mu_k} = (\nu_1, \nu_2, ..., \nu_r). \tag{2.6}$$

The expected frequency of a single outcome A_i is $\mathrm{E}(\nu_i) = np_i$, with a variance $\mathrm{var}(\nu_i) = np_i(1 - p_i)$. This matches that of the binomial distribution, and we might therefore be fooled into thinking that the vector of frequencies $\boldsymbol{\nu}$ is binomially distributed. This is *not* the case, as the individual frequencies are *dependent* random variables with a restrained sum equal to n. Instead, the vector of frequencies $\boldsymbol{\nu}$ has a *multinomial* distribution, with parameters $n > 0$ and $\boldsymbol{p} = (p_1, p_2, ..., p_k)$, where $0 \leq p_i \leq 1$ and $\sum p_i = 1$.

The probability mass function (PMF) of the multinomial distribution is given by

$$\mathrm{Prob}(\boldsymbol{\nu} = \boldsymbol{x}) = \mathrm{Prob}(\nu_1 = x_1, \nu_2 = x_2, ..., \nu_r = x_r) \tag{2.7}$$

$$= \frac{n!}{x_1!...x_r!} p_1^{x_1}...p_k^{x_k}, \tag{2.8}$$

where $\boldsymbol{x} = (x_1, ..., x_r)$ is any vector of integers with $0 \leq x_i \leq n$ and $\sum_{i=1}^{r} x_i = n$. The vector of expected frequencies is

$$\mathrm{E}(\boldsymbol{\nu}) = n\boldsymbol{p} = (np_1, np_2, ..., np_r). \tag{2.9}$$

In the special case of the multinomial distribution where all the $p_i = p = \frac{1}{r}$ are equal, the PMF can be written as

$$\text{Prob}(\boldsymbol{\nu} = \boldsymbol{x}) = \frac{n!}{x_1!...x_r!}\, p^{x_1+...+x_k} = \frac{n!}{x_1!...x_r!}\, p^n, \tag{2.10}$$

and the vector of expected frequencies becomes

$$\text{E}(\boldsymbol{\nu}) = n\boldsymbol{p} = (np, np, ..., np). \tag{2.11}$$

To test whether the observed frequencies ν_i match the assumed multinomial distribution, we can define the null hypothesis

$$H_0 : \; p_i = p_i^{(0)}, \quad i = 1, 2, ..., r. \tag{2.12}$$

We can then use Pearson's chi-squared test. Pearson's test statistic is defined as

$$\begin{aligned} X^2 &= \sum_{i=1}^{r} \frac{\left(\nu_i - np_i^{(0)}\right)^2}{np_i^{(0)}} = \sum_{i=1}^{r} \frac{\nu_i^2 - 2\nu_i np_i^{(0)} + \left(np_i^{(0)}\right)^2}{np_i^{(0)}} && (2.13) \\ &= \sum_{i=1}^{r} \frac{\nu_i^2}{np_i^{(0)}} - 2\sum_{i=1}^{r} \nu_i + n\sum_{i=1}^{r} p_i^{(0)} = \sum_{i=1}^{r} \frac{\nu_i^2}{np_i^{(0)}} - 2n + n && (2.14) \\ &= \frac{1}{n}\sum_{i=1}^{r} \frac{\nu_i^2}{p_i^{(0)}} - n. && (2.15) \end{aligned}$$

For the special case where all the $p_i^{(0)} = p^{(0)} = \frac{1}{r}$ are equal, this simplifies further to

$$X^2 = \frac{r}{n}\sum_{i=1}^{r} \nu_i^2 - n. \tag{2.16}$$

When the sample size is large, this statistic has an asymptotic chi-squared (χ^2) distribution with $r-1$ degrees of freedom. The reduction of 1 degree of freedom is because there is one constraint, namely that the frequencies have to sum to n. If the expected frequencies $np_i^{(0)}$ are too small (typically < 5) the chi-squared distribution will not be a good approximation to the distribution of the test statistic. The individual trials are assumed to be independent.

Knowing the distribution, the p-value, defined as the probability of finding a test statistic as large or larger than the observed, i.e., $\text{Prob}(\chi^2 \geq X^2)$, can be calculated. A perfect fit, i.e., $\nu_i = np_i^{(0)}$ for all i, will give a $X^2 = 0$, making the p-value 1. A poor fit will give a large X^2, and therefore p-value close to (but never equal to) 0. Thus, the p-value from a chi-squared test can be seen as a measure of the goodness-of-fit of the data to the expected distribution, a property we will later exploit.

2.2 The Kolmogorov-Smirnov test

The following section is based on *Advanced statistics from an elementary point of view* (Panik 2005). Pearson's chi-squared test is well suited for analyzing categorical data or data that naturally fall into distinct groups or bins. To use the chi-squared test on *continuous* data, we would have to group the data into somewhat arbitrarily sized bins. Because the expected frequencies of these bins cannot be too low, there is a lower limit to the bin sizes, and an upper limit to the number of bins. This necessarily causes information to be lost. The chi-squared test is therefore not ideal for continuous data. The Kolmogorov-Smirnov (KS) test is a GOF test that does not require grouping of the sample data, and is therefore much better suited for continuous data.

Let X be a random variable, and $x_i,\ i = 1, 2, ..., n$ be an *ordered* set of n realizations of X. We wish to determine if X has a specific hypothesized distribution, or, in other words, if the observed data x_i stems from a population with a cumulative distribution function (CDF) $F(x)$ that equals our hypothesized CDF $F_0(x)$. We formulate our null hypothesis,

$$H_0 : F(x) = F_0(x). \tag{2.17}$$

For a *two-sided* KS test, the alternative hypothesis becomes

$$H_1 : F(x) \neq F_0(x). \tag{2.18}$$

Our sample data will have an *empirical distribution function* (EDF) $S_n(x)$ that converges to the true CDF $F(x)$ for large n. $S_{\mathrm{n}}(x)$ equals the proportion of realizations x_i that are below x. It is therefore a discrete function, increasing stepwise by $1/n$ at each $x = x_i$. It can be defined as

$$S_n(x) = \begin{cases} 0, & \text{for } x < x_1 \\ \frac{i}{n}, & \text{for } x_i \leq x < x_{i+1},\ i = 1, 2, ..., n-1 \\ 1, & \text{for } x \geq x_n. \end{cases} \tag{2.19}$$

The Kolmogorov-Smirnov test statistic is defined as the *supremum* of the absolute difference between $S_n(x)$ and $F_0(x)$ for all x,

$$D_n = \sup_x |S_n(x) - F_0(x)|, \tag{2.20}$$

or simply the greatest distance between the two. If H_0 is true, we expect D_n to be small. The sampling distribution of D_n is known, and we can compare our value with this distribution and calculate a p-value.

As an example of the usage of the KS test we can test the output of a PRNG, a series of numbers in the half-open interval $[0, 1)$, against the expected

uniform distribution. Then, under H_0, $X \sim \mathcal{U}(0,1)$. The expected CDF becomes

$$F_0(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } 0 \leq x \leq 1 \\ 1, & \text{for } x > 1. \end{cases} \tag{2.21}$$

Having obtained a set of pseudorandom numbers x_i, the first step is to order these values in increasing order. The EDF can then be found from the definition of S_n in Equation 2.19, and the test statistic D_n is found by Equation 2.20. In Figure 2.1 the red line marks the greatest distance $D_n = 0.11$ between $S_n(x)$ and $F_0(x)$ for a set of 100 pseudorandom numbers supposedly drawn from $\mathcal{U}(0,1)$.

The p-value, i.e., the probability of observing a more extreme value of the KS test statistic for $n = 100$, can be shown to be 0.16. There is in other words no evidence to support a rejection of the null hypothesis that the x_i are drawn from $\mathcal{U}(0,1)$ at any meaningful level of significance α.

Unlike the chi-squared test, where the test statistic only approximates the chi-squared distribution, the KS test is an exact test, meaning it can be used for small n as well as large.

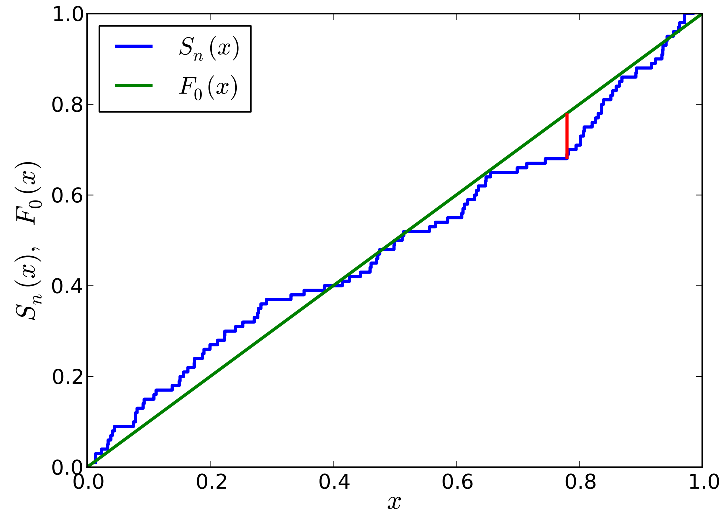


Figure 2.1: Empirical distribution function $S_n(x)$ (blue line) of $n = 100$ pseudorandom numbers from PRNG, supposedly drawn from a uniform distribution, and the cumulative distribution function $F_0(x)$ (green line) expected under H_0. The red line marks the KS test statistic D_n, the greatest distance between the two lines.

2.3 The Z-test

The following section is based on *Advanced statistics from an elementary point of view* (Panik 2005).

The Z-test is a test of the hypothesis that a sample is drawn from a normal population with the mean μ_0. Let $(X_1, X_2, ..., X_n)$ be a set of n random variables, drawn from a normal population with a known variance σ^2, but an unknown population mean μ. The sample mean is denoted $\bar{X}$, and the standard deviation of the mean is related to the standard deviation of the population by $\sigma_{\bar{X}} = \sigma/\sqrt{n}$.

We wish to determine whether the population mean μ equals a hypothesized mean μ_0, i.e.,

$$H_0 : \mu = \mu_0. \tag{2.22}$$

We consider here only the two-sided alternative hypothesis,

$$H_1 : \mu \neq \mu_0. \tag{2.23}$$

The test statistic used is the standard score,

$$Z = \frac{\bar{X} - \mu_0}{\sigma_{\bar{X}}} \tag{2.24}$$

which, under H_0, has a standard normal distribution $\mathcal{N}(0, 1)$. The p-value of the two-sided Z-test is the probability of finding a value of Z as extreme or more extreme than the observed value z under H_0, i.e.,

$$\begin{aligned} p\text{-value} &= \text{Prob}(Z \leq -|z|) + \text{Prob}(Z \geq |z|) & (2.25) \\ &= 2\text{Prob}(Z \geq |z|)\,. & (2.26) \end{aligned}$$

If the data set consists of a single random variable X, the test statistic becomes

$$Z = \frac{X - \mu_0}{\sigma}. \tag{2.27}$$

The Z-test can only be used for data that can be approximated by a normal distribution. The central limit theorem is typically invoked to justify the approximation.

Chapter 3

Distribution of connections

We will now describe a procedure for testing random convergent and divergent connection algorithms for networks without spatial structure. These connection algorithms were described in the introduction, and an example of a resulting network was shown in Figure 1.1.

3.1 Random convergent connections

For each target node `RandomConvergentConnect` iterates over, it randomly draws C source nodes from the $r = N_\mathrm{s}$ available source nodes and connects to them. Multapses and autapses are allowed, i.e., nodes are drawn with replacement. We are interested in checking whether all the source nodes are drawn with equal probability. In other words, we want to test the observed distribution of connections against the expected uniform distribution. Each drawing of a source node can be thought of as a trial as described in Section 2.1, with exactly *one* outcome A_i. The total number of trials is $n = N_\mathrm{t} \times C$, where N_t is the number of target neurons. After all n trials, each source node i will have some out-degree (number of outgoing connections), described by the *frequency of outcome* A_i as defined in Equation 2.5:

$$\nu_i = \sum_{k=1}^{n} \mu_{ki}. \tag{3.1}$$

The vector of frequencies,

$$\boldsymbol{\nu} = \sum_{k=1}^{n} \boldsymbol{\mu_k} = (\nu_1, \nu_2, ..., \nu_r), \tag{3.2}$$

now contains all observed out-degrees.

In this particular case, the *expected* frequencies are all the same, $\mathrm{E}(\nu_i) = np^{(0)}$, so the vector of expected frequencies contains r equal entries, $\mathrm{E}(\boldsymbol{\nu}) = n\boldsymbol{p}^{(0)} = (np^{(0)}, np^{(0)}, ..., np^{(0)})$.

To test the hypothesis

$$H_0 : p_i = p_i^{(0)} = p^{(0)} = \frac{1}{r}, \quad i = 1, 2, ..., r, \tag{3.3}$$

i.e., that source nodes are drawn with equal probability, we use Pearson's chi-squared (χ^2) test. As explained in Section 2.1, the statistic becomes

$$X^2 = \frac{r}{n} \sum_{i=1}^{r} v_i^2 - n. \tag{3.4}$$

This statistic has an asymptotic chi-squared (χ^2) distribution with $r-1$ degrees of freedom. Knowing the distribution, the p-value can be calculated.

Instead of simply rejecting or accepting H_0 based on one p-value, it is, as argued earlier, better to apply a two-level test, i.e., test multiple p-values for a uniform distribution. This assumes the p-value is uniformly distributed under H_0, which is not strictly true for p-values coming from a chi-squared test, as X^2 is discrete. To examine the discreteness of X^2 closer, let us change ν_1 to $\nu_1 + 1$, and ν_2 to $\nu_2 - 1$. The resulting change in X^2 is

$$\begin{aligned}
\Delta X^2 &= \left[\frac{r}{n}\left([\nu_1 + 1]^2 + [\nu_2 - 1]^2 + \sum_{i=3}^{r} \nu_i^2\right) - n\right] - \left[\frac{r}{n}\sum_{i=1}^{r} \nu_i^2 - n\right] \\
&= \frac{r}{n}\left([\nu_1 + 1]^2 + [\nu_2 - 1]^2 - \nu_1^2 - \nu_2^2\right) \\
&= \frac{2r}{n}(\nu_2 - \nu_1 - 1)
\end{aligned}$$

Thus, the smallest non-zero difference between two possible values of X^2 is $2r/n = 2N_\mathrm{s}/(N_\mathrm{t}C)$. As long as $n = N_\mathrm{t}C$ is large enough compared to $r = N_\mathrm{s}$, therefore, these "jumps" in X^2 are small, and we may treat it as continuous; the effects of small $N_\mathrm{t}C$ is investigated in Section 3.1.2. This means that the two-sided Kolmogorov-Smirnov (KS) test can be used to test the uniformity of the p-values. The KS test produces a p-value which, if it is smaller than a chosen significance level α, leads us to reject H_0. An advantage of this approach is that, even though Pearson's chi-squared test is one-tailed, a "too good" fit (connections are more evenly distributed than is likely to happen by chance) will be detected, as there will be an excess of large p-values from the chi-squared tests.

3.1.1 Implementation

The test procedure outlined above is implemented as a Python module, included in Appendix B. The module defines a class, `RCC_tester`. The class has two methods, `chi_squared_test` and `two_level_test`, for testing the connections created by `RandomConvergentConnect`.

`chi_squared_test` creates two sets of nodes, `source_nodes` and `target_nodes`, and connects them using `RandomConvergentConnect`. It then runs a chi-squared test on the out-degrees of the source nodes, and returns the test statistic and the p-value. If the expected frequencies np_i are too small, results may be unreliable. Thus, if they are smaller than e_{min} (10 by default), a warning is displayed. e_{min} can be changed. The chi-squared test is implemented using the `chisquare` function from the `scipy.stats` library.

The method `two_level_test` runs `chi_squared_test` n_{runs} times, and checks the returned p-values for uniformity using the two-sided KS test. The resulting KS test statistic and p-value is returned. The KS test is implemented using the `kstest` function from the `scipy.stats` library. In the main section at the end of the module an example of how to use the `RCC_tester` class is provided.

Between each run of `RandomConvergentConnect`, the network is deleted by calling the function `ResetKernel`, to avoid memory bloat. This also resets the PRNGs. Thus, for each run of `RandomConvergentConnect`, NEST must be given a new set of PRNG seed values. NEST requires one seed value for the global PRNG and one for each per-process PRNG, totaling $1 + n_{\text{vp}}$ seed values, where n_{vp} is the number og virtual processes (VPs) used by NEST. The `chi_squared_test` method takes one argument, a "master seed" `msd`, and the $1+n_{\text{vp}}$ PRNGs are seeded with the values (`msd`, `msd`+1, ..., `msd`+n_{vp}). For independent results, `chi_squared_test` should be given a new master seed for each run, differing by at least $n_{\text{vp}} + 1$. The `two_level_test` method handles this automatically when running `chi_squared_test`. The first of the master seeds can be passed as an argument `start_seed`. When running `two_level_test` multiple times, `start_seed` should differ by at least $n_{\text{runs}}(n_{\text{vp}} + 1)$.

3.1.2 Results

Running the script in in Appendix B with the parameter set shown in Table 3.1, we obtain a set of chi-squared p-values with the distribution shown in Figure 3.1. The KS test of the uniformity of these p-values results in the KS

Parameter	N_{s}	N_{t}	C	n_{runs}	`start_seed`
Value	1,000	1,000	1,000	10,000	0

Table 3.1: Default parameter set for reported results.

test statistic $D_n = 5.67 \times 10^{-3}$ and a p-value of 0.905, leading us to accept the null hypothesis H_0 (defined in Equation 3.3) that the nodes were selected with equal probability.

To assess the two-level testing procedure itself, it might be useful to be able to run it on data that we can safely assume to fulfill H_0. The test

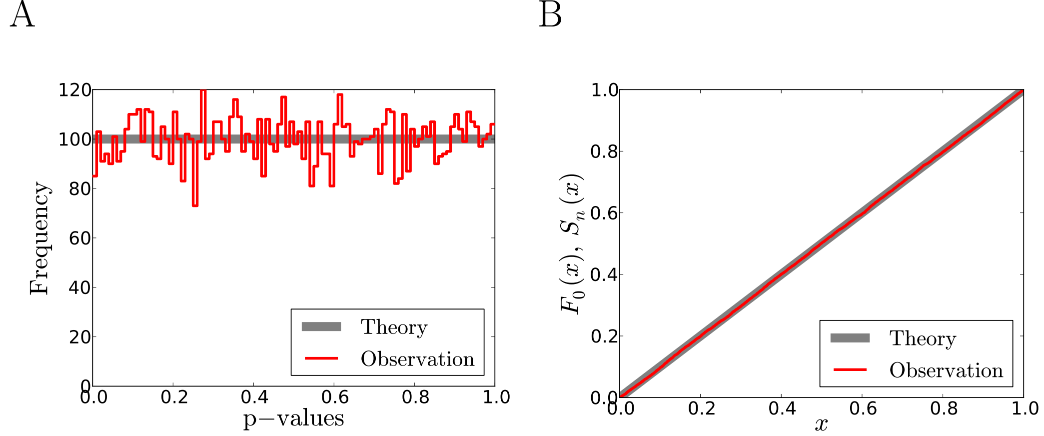


Figure 3.1: A: Histogram of p-values from chi-squared goodness-of-fit tests of $n = 10{,}000$ individual networks, combined into 100 bins (red) together with the expected uniform distribution (grey). B: The corresponding empirical distribution $S_n(x)$ (red) and expected cumulative distribution $F_0(x)$ (grey).

script allows us to do this with the optional argument `control=True` passed to `two_level_test`. NEST's connection algorithm is then swapped with a simple "control algorithm" that creates data that matches the multinomially distributed vector of degrees we would expect to get from NEST. It runs faster than the actual connection algorithm, and can therefore generate a larger data set in the same period of time. As mentioned briefly in Section 3.1, the p-value from a chi-squared test is not truly a continuous variable. This distinctness might cause the two-level test to give a left-skewed distribution of p-values for certain combinations of parameters. The control algorithm can be used to investigate this. Running the two-level test procedure on the control algorithm, with $n_{\text{runs}} = 1{,}000$, repeated 100 times, each time with a different starting seed, yields the EDF of p-values in Figure 3.2. These p-values appear to be uniformly distributed (a KS test of uniformity results in the p-value 0.468). This is an important result, as the uniformity of the p-values from the two-level test procedure under H_0 is a prerequisite for drawing conclusions based on them.

The jumps of the chi-squared p-value will be large when the expected out-degree $N_{\text{t}}C/N_{\text{s}}$ is small, i.e., for small N_{t} and C, and large N_{s}. The effect of small degrees is investigated. Figure 3.3 shows the EDF of p-values with four different combinations of small values for these parameters. Jumps in the p-values are clearly seen. As expected, the jumps grow larger with a smaller expected out-degree. The p-values from the two-level test corresponding to the EDFs in Figure 3.3A, 3.3B, and 3.3C, are 2.47×10^{-25}, 6.47×10^{-5}, and 0.0162, respectively. These p-values are clearly left-skewed, even though the data tested does fulfill H_0. In the last figure, 3.3D, the expected degree is 100,

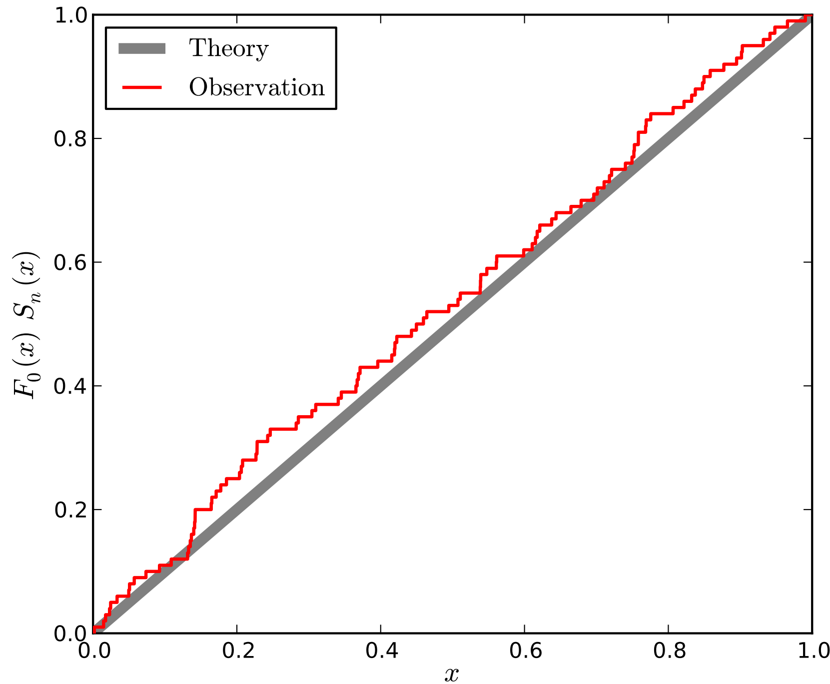


Figure 3.2: EDF of 100 p-values from two-level test procedure. For each run, the two-level test procedure generates $n = 1{,}000$ networks using the control algorithm, runs a chi-squared GOF tests on the vector of degrees, and tests the resulting p-values for uniformity using the KS test.

and the p-value resulting from the two-level test is 0.237. Re-running multiple times reveals that the fraction of p-values below α is close to α. Thus, an expected degree of about 100 seems to suffice to produce p-values that can be used to draw meaningful conclusions.

Note that even though the two-level test will result in left-skewed p-values for small expected degrees, the fraction of p-values from a chi-squared test that lie below some level of significance α will be very close to α, partly because the biggest jumps of the EDF are in the middle region, while in the lower part, close to 0, the curves are fairly smooth (see Figure 3.3). In other words, a two-level test might not be reliable for small expected degrees ($\lesssim 100$), but a "one-level" chi-squared test can be used with much smaller expected degrees (~ 5).

Sensitivity

To assess the sensitivity of the two-level test procedure, i.e., its ability to detect small errors and biases, various errors and biases were deliberately introduced into the data. Some of these are described below, as well as the resulting p-values. We emphasize that these p-values are only examples. Rerunning the

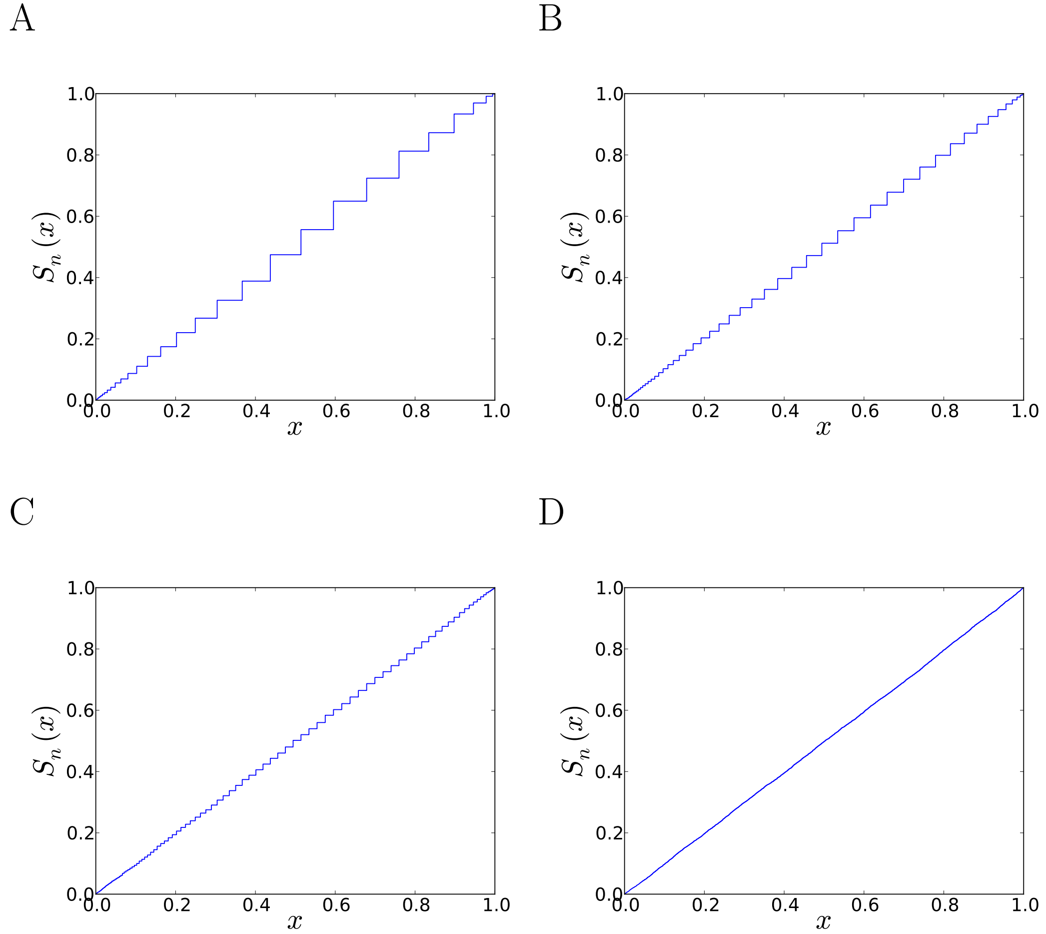


Figure 3.3: EDFs of 10,000 p-values from chi-squared tests with four different combinations of values for the parameters N_s, N_t, and C. In A, $N_s = 10$, $N_t = 5$, and $C = 5$, resulting in an expected out-degree of 2.5. The p-values are clearly distinct, with fairly large jumps between adjacent values, especially for medium to large values. In B, $N_s = 10$, $N_t = 10$, and $C = 5$. This gives an expected degree of 5, resulting in smaller jumps. In C, $N_s = N_t = C = 10$, resulting in an expected degree of 10, and the jumps are smaller yet. In D, $N_s = 10$, $N_t = 100$, and $C = 10$, giving an expected degree of 100. The jumps are no longer visible.

tests with the same biased algorithm and the same parameters, but with a different PRNG seed value, will result in a different p-value, possibly one that differs by quite a bit. Thus, they are only meant to give an indication of how well the bias is detected. Unless otherwise stated, the parameter values listed in Table 3.2 are used.

The first bias that was deliberately introduced was a simple right-skewing of the data. The degree of the first half of the nodes was reduced by one, and the degrees of the second half was increased by one. This was easily detected by the test procedure ($p = 1.71 \times 10^{-5}$). In Figure 3.4, the distribution of 1,000

Parameter	N_s	N_t	C	n_{runs}	start_seed	control
Value	1,000	1,000	100	1,000	0	True

Table 3.2: Default parameter set for sensitivity testing.

p-values are shown. It is clear from the figure that a single chi-squared test would not in any consistent way detect the bias. Only when we accumulate a large number of p-values is the trend obvious.

When C was increased to 1,000, the bias was not detected ($p = 0.255$). A large C is clearly not always advantageous for detecting small biases, especially biases that do not increase with C. Only after n_{runs} was increased to 10,000 was the bias again detected ($p = 0.00428$).

A second bias was introduced by decreasing the degree of every source node with an even-numbered index by one, and increasing the degree of every source node with an odd-numbered index by one. This was also easily detected ($p = 7.13 \times 10^{-6}$).

A third bias was caused by increasing by one the degrees below the 5th percentile, and decreasing by one the degrees above the 95th percentile, thereby making the vector of frequencies a slightly "too good" fit to the theoretical distribution. This change was detected, both with $C = 100$ ($p = 8.02 \times 10^{-14}$) and $C = 1{,}000$ ($p = 0.00231$), for $n_{runs} = 100$, as well as for larger n_{runs}.

A fourth bias was introduced by moving all the connections from one of the N_s nodes to another node. This kind of error could easily be introduced by confusion between the zero-based numbering typically used in computer science and the one-based numbering used in everyday circumstances. The

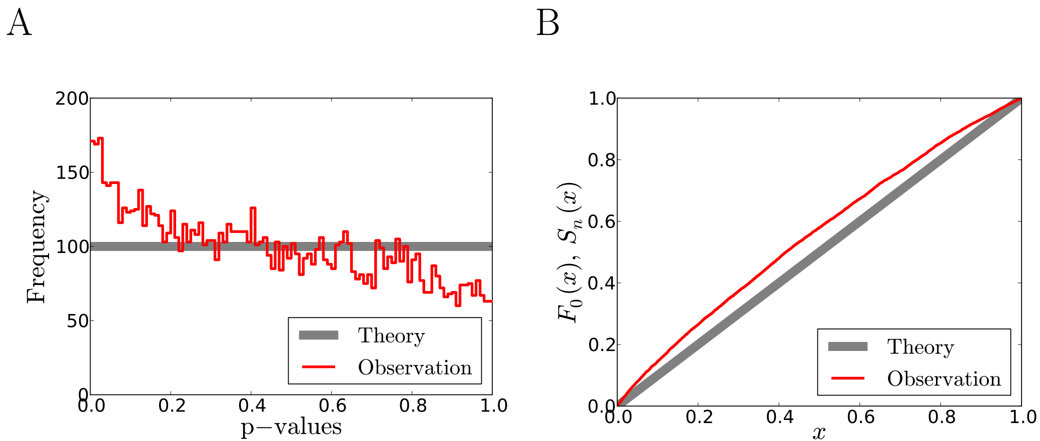


Figure 3.4: Histogram (A) and CDF (B) showing the distribution of p-values from chi-squared tests after introducing an error into the connection algorithm. Expectation is shown in grey.

error was easily detected (the p-value was reported as 0.0 as it was too small for Python's built-in floating point data type to handle). Increasing N_s to 1,000, the error was still detected at $\alpha = 0.05$ ($p = 0.0366$).

A fifth, very small bias was introduced by increasing the degree of one of the source nodes by one, and decreasing the degree of another node by one. The bias was not detected at $\alpha = 0.05$ with $C = 100$ and $n_{runs} = 10{,}000$ ($p = 0.0740$), nor with C reduced to 10. Reducing N_s and N_t to 100, the bias was detected at $\alpha = 0.05$ ($p = 0.0189$). A smaller network clearly increases the sensitivity to certain types of biases, especially biases that do not increase with network size.

Several similar tests were run. Generally the two-level test procedure seems quite sensitive to small biases. The sensitivity obviously increases with n_{runs}, but not necessarily with C, N_t and N_s. The effect of these parameters on the sensitivity depends on the nature of the bias we wish to detect. It might therefore be a good idea to run the procedure with different sets of parameters.

3.2 Random divergent connections

`RandomDivergentConnect` works in very much the same way as `RandomConvergentConnect`, except that the source nodes are now iterated over, and target nodes are randomly drawn and connected, meaning that the out-degrees of the source nodes are now all C, while the in-degrees of the target nodes will vary. It might therefore seem superfluous to test both functions. Running NEST with multiple VPs, however, the exact implementation of the connection algorithm is not the same for the two functions. This is because different nodes are handled by different VPs, and information about connections is handled by the same VP by which the *target* node is handled (Plesser, Eppler, Morrison, Diesmann, and Gewaltig 2007). To minimize the amount of inter-VP communication necessary, each VP is given its own PRNG. `RandomDivergentConnect` uses the global PRNG when drawing connections, while `RandomConvergentConnect` uses the per-VP PRNGs. The result is different connectivity patterns, even with the same master seed value. Both function should therefore be tested with multiple VPs.

3.2.1 Implementation

The Python module for testing `RandomDivergentConnect`, found in Appendix C, is very similar to the module for testing `RandomConvergentConnect`. A class `RDC_tester`, with the two methods `chi_squared_test` and `two_level_test`, is defined. These methods take the same parameters as their counterparts in the convergent case. As before, the usage is demonstrated in the main section of the module.

3.2.2 Results

Running the script in in Appendix C with the same parameter set used for random convergent connections, listed in Table 3.1, results in a p-value of 0.905. This matches exactly the p-value we got in Section 3.1.2 after running the test script in Appendix B with the same set of parameters. In fact, the distribution of connections among source nodes for a network created by `RandomConvergentConnect` will exactly match the distribution of connections among target nodes for a network created by `RandomDivergentConnect`, when NEST is run with one VP. For multiple VPs, however, the resulting distribution of connections is not the same. The test class in the test script can be instantiated with the additional argument `threads`, causing NEST to operate with the specified number of local threads. With NEST running as a single process, the number of VPs will equal the number of local threads. Running the script in Appendix C with the same parameter set as before (listed in Table 3.1), but with the extra argument `threads = 2`, the we obtained a p-value of 0.931, consistent with the expected multinomial distribution of in-degrees.

3.3 Automated test procedure

We will now describe how to turn a variant of the test procedure described in the previous sections into an automated test, implemented as a unit test. This places some practical limitations on the test procedure. The test must not take too long to run. It should also not have a too high rate of false positives (type I error). At the same time we do of course not want to loose too much sensitivity.

A quick and efficient approach would be to generate one single network and run a chi-squared test on the distribution of connections with some relatively low level of significance α, say, 0.01, as described earlier, but even with this low level of significance, we would still see false positives for about 1% of the test runs. Reducing α further would result in a lower sensitivity. On the other hand, running the more thorough but slow two-level test procedure described in Section 3.1 every time is quite time-consuming. Instead, an adaptive approach is proposed here, similar to the one used by L'Ecuyer and Simard (2007). First, a single network is generated, and a chi-squared test is performed on the distribution of connections. If the resulting p-value is deemed too extreme, a larger number n_{runs} of networks is generated, and a more thorough two-level test is performed, either confirming or allaying our suspicion. This will allow us to require a fairly high level of significance for the chi-squared test, thereby keeping the sensitivity high, while at the same time having a low rate of false positives.

As was shown in Section 3.1.2, not all errors and biases are best detected

with a large network or with large degrees, as this might hide certain types of small biases. Tests should therefore be run with different values of N_s, N_t, and C. For a single chi-square test, the network can be quite small, and the only limit is that the expected degree must not be below e_{min}, but the two-level test procedure is more sensitive, and the expected degree should not be much lower than 100. Both `RandomConvergentConnect` and `RandomDivergentConnect` should be tested. Running tests both with a single VP and with multiple VPs might also be a good idea, as the implementation of the connection algorithms are somewhat different for multiple VPs (see Section 3.2).

For the p-values from the chi-squared test, both very low and very high values are considered suspicious. Let $\alpha_{1,\text{lower}}$ and $\alpha_{1,\text{upper}}$ be the values below and above which, respectively, p-values are deemed suspicious. Under H_0, suspicious values will occur for a fraction $\alpha_1 = \alpha_{1,\text{lower}} + (1 - \alpha_{1,\text{upper}})$ of the tests performed. Whenever such an extreme value is encountered, the two-level test is performed. Let α_2 be the value below which the p-value from the KS test is considered too extreme, and H_0 is rejected. The total fraction of false positives will then be

$$\alpha = \alpha_1 \times \alpha_2 = (\alpha_{1,\text{lower}} + 1 - \alpha_{1,\text{upper}}) \times \alpha_2 \tag{3.5}$$

With $\alpha_{1,\text{lower}} = 0.025$, $\alpha_{1,\text{upper}} = 0.975$, and $\alpha_2 = 0.05$, α becomes 0.0025. The choice of these critical values, as well as the test parameters (N_s, N_t, C, n_{runs}) will depend on the intended usage of the unit test, the computing power of the system it will run on, the maximum time the test can be allowed to take, the desired fraction of false positives, etc.

3.3.1 Implementation

An implementation of the automated test procedure can be found in Appendix D. It is implemented using the Python unit testing framework `unittest`. The test scripts for `RandomConvergentConnect` and `RandomDivergentConnect`, found in Appendix B and C, are imported to avoid code duplication.

The test is repeated three times for each of `RandomConvergentConnect` and `RandomDivergentConnect`, once with a small network, once with a larger network, and once with multiple VPs, giving a total of six test cases. n_{runs} can have different values for each test, as we might want to run a test on a small network a larger number of times than a test on a large network. The critical values are $\alpha_{1,\text{lower}} = 0.025$, $\alpha_{1,\text{lower}} = 0.975$, and $\alpha_2 = 0.05$. We thus expect a fraction 0.0025 of the tests to fail. When running the test suite with the six test cases 500 times, giving a total of 3,000 test runs, 10 failures were encountered. This is close to the $\sim 0.0025 \times 3000 = 7.5$ false positives we would expect.

When the six one-level chi-squared tests are passed, the suite takes less than a minute to complete on most systems. In the event that all six one-level

tests fail (which will happen with a probability of $0.05^6 = 1.5625 \times 10^{-8}$ under H_0), and the two-level test is invoked, the entire suite might take a few minutes to complete, depending on hardware, installed version of NEST etc.

Chapter 4

Spatially structured networks

We will now describe the procedure for testing the probabilistic connection algorithms for spatially structured networks. We briefly discussed spatially structured networks in the introduction, and an example was show in Figure 1.2. In NEST, these connection patterns can be created between two- or three-dimensional layers using the function `ConnectLayers` from the `Topology` module (Plesser and Enger 2012).

4.1 Two-dimensional space

Concepts and derivations in this section are based on Kriener (2012). Let one node be centered on a quadratic $L \times L$ layer. It can connect to N other nodes, uniformly distributed on the layer, as long as they are inside the mask. The mask has the same size as the layer, and for now, the same position. The probability of making a connection is given by a distance-dependent connection probability (kernel). Let $(x_i)_m$ be the mth component of the coordinate vector for node i. Let

$$D_{ij} = \sqrt{\sum_{m=1}^{k} (\Delta x_{ij})_m^2}, \tag{4.1}$$

where $k = 2$ is the number of dimensions, be the distance between nodes i and j. With periodic boundary conditions,

$$(\Delta x_{ij})_m = \begin{cases} |(x_i)_m - (x_j)_m| & \text{for } |(x_i)_m - (x_j)_m| \leq L/2 \\ L - |(x_i)_m - (x_j)_m| & \text{for } |(x_i)_m - (x_j)_m| > L/2 \end{cases} \tag{4.2}$$

We wish to compare the observed distribution of distances between connected nodes with the expected distribution. The expected distribution depends on both on the number of potential targets at a given distance, and the probability of connecting to nodes at that distance (given by the kernel). We

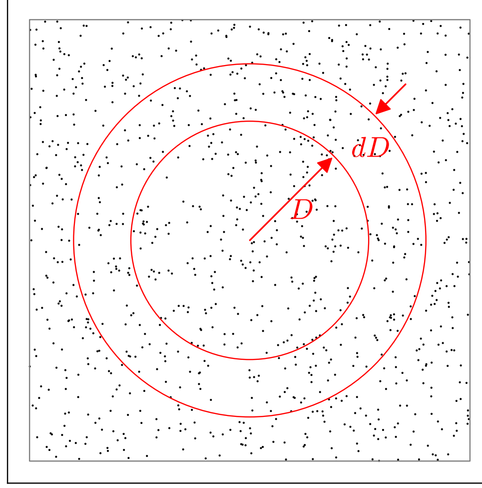


Figure 4.1: A ring of radius D and thickness $\mathrm{d}D$ is placed on a layer with a node density $\rho_0 = N/L^2$. The expected number of nodes inside the ring is then $\mathrm{d}N = \rho_0\pi\left[(D+\mathrm{d}D)^2 - D^2\right]$.

start by deriving an expression for the number of potential targets at a given distance.

Let $\rho_0 = N/L^2$ be the average node density on the layer. Now consider a ring like the one in Figure 4.1, with radius D and thickness $\mathrm{d}D$. The expected number of nodes in the ring will be

$$\mathrm{d}N = \rho_0\pi\left[(D+\mathrm{d}D)^2 - D^2\right] \tag{4.3}$$

and the density of nodes in the ring is given by $\mathrm{d}N/\mathrm{d}D$. We now define the radial distribution function (RDF) as the limit of the node density for an infinitesimally thin ring

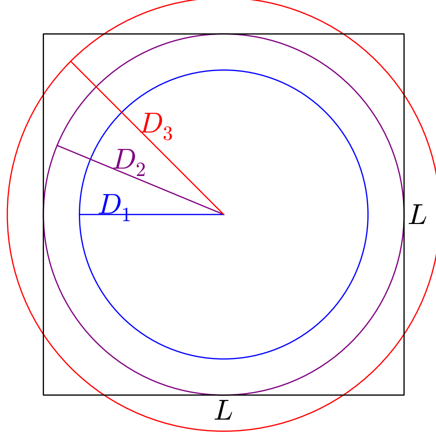
$$\rho(D) = \lim_{\mathrm{d}D\to 0}\frac{\mathrm{d}N}{\mathrm{d}D} = 2\pi\rho_0 D\ . \tag{4.4}$$

Thus, for distances $D \in [0, L/2]$ from the center node, the RDF is proportional to the circumference of a circle with radius D. For $D \in (L/2, L/\sqrt{2}]$, the mask comes into play, as part of the circle of radius D is outside the mask. This is illustrated in Figure 4.2A. As seen in Figure 4.2B, only a fraction $2\beta/\pi$ lies inside the mask, where

$$\beta = \frac{\pi}{2} - 2\alpha = \frac{\pi}{2} - 2\arccos\left(\frac{L}{2D}\right). \tag{4.5}$$

Substituting for β from 4.5 the fraction becomes

$$\frac{2\beta}{\pi} = \frac{\pi - 4\arccos\left(\frac{L}{2D}\right)}{\pi}. \tag{4.6}$$



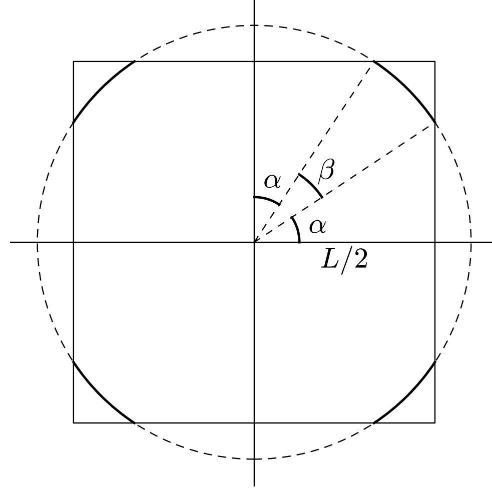


Figure 4.2: A: Illustration of the effect of the mask on the radial distribution function (RDF). At distance D_1, the RDF is $\rho_0 2\pi D$. At distances $D \geq D_2$, the mask comes into play. At distance D_3, for example, part of the circle is outside the mask, and the number of nodes eligible for connecting to thus reduced. B: The fraction of a circle with radius D which is inside the mask is $2\beta/\pi$. Adapted from Kriener (2012).

The RDF can therefore be summarized as

$$\rho(D) = \begin{cases} \rho_0 2\pi D & \text{for } 0 \leq D \leq \frac{L}{2} \\ \rho_0 2D \left(\pi - 4 \arccos\left(\frac{L}{2D}\right)\right) & \text{for } \frac{L}{2} < D \leq \frac{L}{\sqrt{2}} \\ 0 & \text{otherwise.} \end{cases} \tag{4.7}$$

Let $\mathcal{P}(D)$ be a distance-dependent connection probability function (kernel). The probability density function (PDF) $f(D)$ of distances from the centered node and connected nodes is the normalized product of the RDF and the kernel, i.e.,

$$f(D) = \frac{\rho(D)\mathcal{P}(D)}{\int_0^{L/\sqrt{2}} \rho(R)\mathcal{P}(R)\,\mathrm{d}R}, \tag{4.8}$$

where the denominator is a normalizing constant[1]. The cumulative distribution function (CDF) $F(D)$ can be obtained by integrating $f(D)$, i.e.,

$$F(D) = \frac{\int_0^{D} \rho(R)\mathcal{P}(R)\,\mathrm{d}R}{\int_0^{L/\sqrt{2}} \rho(R)\mathcal{P}(R)\,\mathrm{d}R}. \tag{4.9}$$

[1] R is used instead of D in integrals as D sometimes plays the role of the upper integration limit, for instance in Equation 4.9.

As an example, let $L = 1$ and the kernel be a linear function of D, $\mathcal{P}(D) = (c - aD)H(c/a - D)$, where H is the Heaviside step function. For simplicity we assume that $c/a < L/2$ so that there are no boundary effects. The PDF then becomes

$$f(D) = \frac{\rho_0 2\pi D(c - aD)}{\int_0^{c/a} \rho_0 2\pi R(c - aR)\,\mathrm{d}R} = \frac{6a^2 D(c - aD)}{c^3}, \tag{4.10}$$

and the CDF becomes

$$F(D) = \frac{6a^2}{c^3} \int_0^D R(c - aR)\,\mathrm{d}R = \frac{a^2 D^2 (3c - 2aD)}{c^3}. \tag{4.11}$$

Using numerical integration, CDFs can be similarly found for other kernels. In Figure 4.3, an exemplary PDF and the corresponding CDF are shown, as well as the connectivity pattern of the network. A Gaussian kernel is used.

Using the two-sided KS test, the observed EDFs can now be compared with these theoretical CDFs, with the null hypothesis H_0 that distances are drawn from the theoretical PDFs. The KS test is the natural choice here because we are looking at the distribution of a continuous parameter, the distance.

As will be discussed in Section 4.1.2, a few types of errors will not be detected by the KS test of the distribution of source-target distances. A second test is therefore implemented. It simply compares the total number of connections C with the expected number of connections C_0. C will be the sum of N independent Bernoulli random variables with different success probabilities p given by the kernel. Its distribution is the Poisson binomial distribution, a generalization of the binomial distribution that does not require all success probabilities to be the same (Wang 1993). For large N it approximates the normal distribution $\mathcal{N}(\mu, \sigma^2)$, with $\mu = \sum_{i=1}^N p_i$ and $\sigma^2 = \sum_{i=1}^N p_i(1 - p_i)$. The Z-test can thus be used as described in Section 2.3. Using the test statistic

$$Z = \frac{C - C_0}{\sigma} = \frac{C - \sum_{i=1}^N p_i}{\sqrt{\sum_{i=1}^N p_i(1 - p_i)}}, \tag{4.12}$$

the two-sided p-value $2\mathrm{Prob}(Z \geq |z|)$ can be calculated.

4.1.1 Implementation

An implementation of the test procedure described above is found in Appendix E. The main block at the bottom demonstrates the usage. The class `ConnectLayers2D_tester` is first instantiated with the required arguments `L` (side length of the square layer), `N` (number of nodes), and `kernel_name` (name of the kernel to use, “constant”, “linear”, “exponential”, or “gaussian”). An optional argument, `kernel_params`, can be used to specify the parameters of the

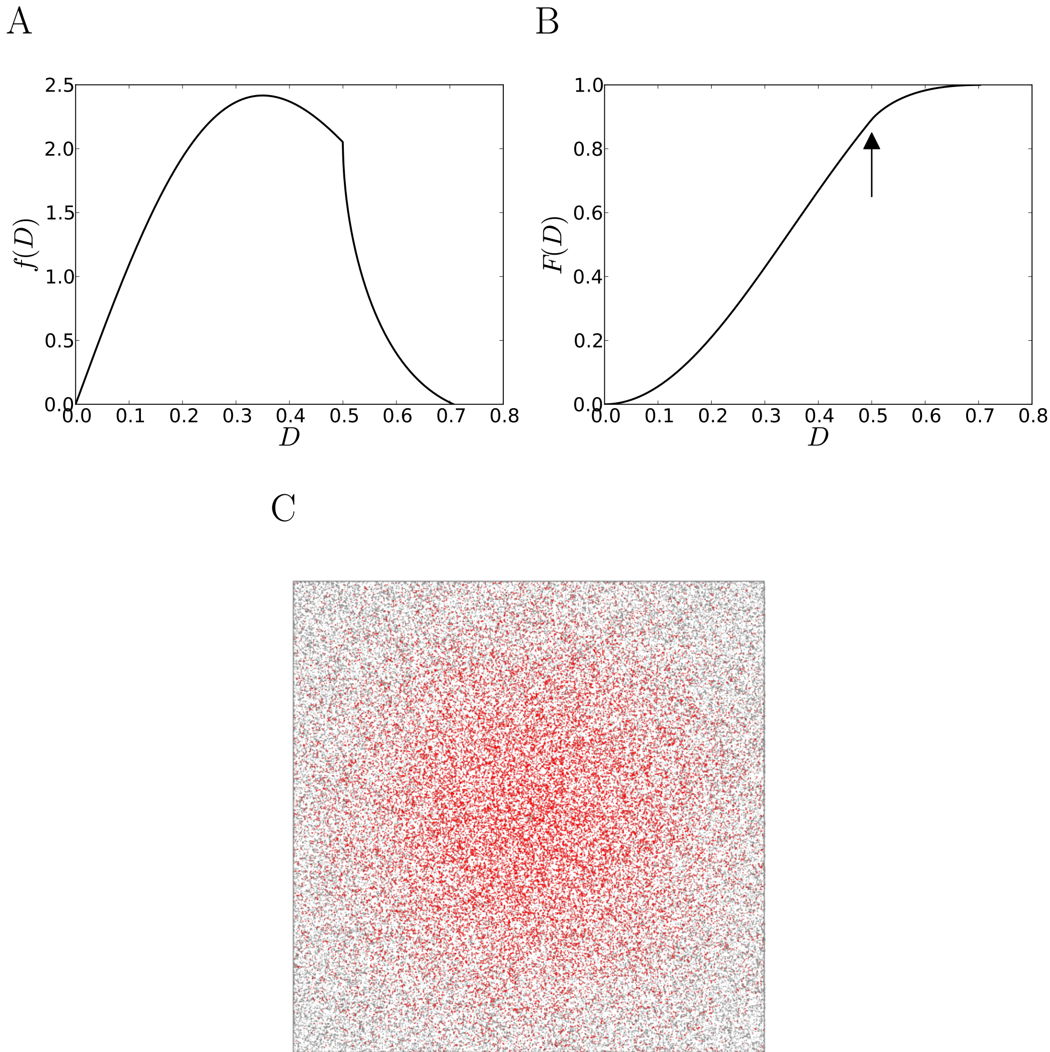


Figure 4.3: Exemplary PDF (A), CDF (B) and connectivity pattern (C). A Gaussian kernel is used, and the layer and mask size is $L^2 = 1$. The sharp kink at $D = 0.5$ in the PDF is due to the mask. A kink exists in the CDF as well, marked by the arrow.

chosen kernel. Without this argument, sensible default values are used. The master seed value can be set using the optional argument `msd`. The value of `msd` is used to seed the PRNG used to draw the uniform, random positions for the nodes. NEST's global PRNG and each of the per-process PRNGs are seeded with the values (`msd` + 1, `msd` + 2, ..., `msd` + $n_{\text{vp}} + 1$), where n_{vp} is the number of virtual processes (VPs). Thus, for independent results, `msd` should differ by at least $2 + n_{\text{vp}}$ between each instantiation of the class. The position of the source node is (0, 0) by default, but it can be changed with the optional `source_pos` argument. The mask is always centered around the source node, regardless of its position.

After the test object is created, the KS test can be run using the `ks_test` method. The KS test statistic and p-value is returned. The PDF is calculated on-the-fly as the product of the relevant kernel and the RDF. The CDF is then found by numerical integration, using the `quad` function from the `scipy.integrate` module. For finite integration limits the function uses the Clenshaw-Curtis method of numerical integration.

The method `z_test` implements the Z-test of the total connection count described earlier. The standard score and the two-sided p-value is returned.

4.1.2 Results

The test script in Appendix E will now be used to test the connection algorithms for connecting 2D layers of spatially structured networks in NEST.

With $L = 1$, $N = 1{,}000{,}000$, and `msd` $= 0$, and with the constant kernel selected, the distribution of distances to connected nodes is as shown in Figure 4.4. There seems to be close agreement between the expected and observed distributions. According to the KS test, there is no evidence to reject the null hypothesis that the distances are drawn from the theoretical PDF ($p = 0.321$). Similar results are found for the linear ($p = 0.837$), exponential ($p = 0.630$) and Gaussian ($p = 0.852$) kernels.

By positioning the source node away from the center, while leaving the mask centered around the node, with the same $L \times L$ size, the periodic boundary conditions come into play. The distribution of source-target distances should not change. An example can be seen in Figure 4.5. With the same parameters as above, but with the source node located at $(L/4,\ L/4)$, the p-values become 0.253, 0.091, 0.004, and 0.113 for constant, linear, exponen-

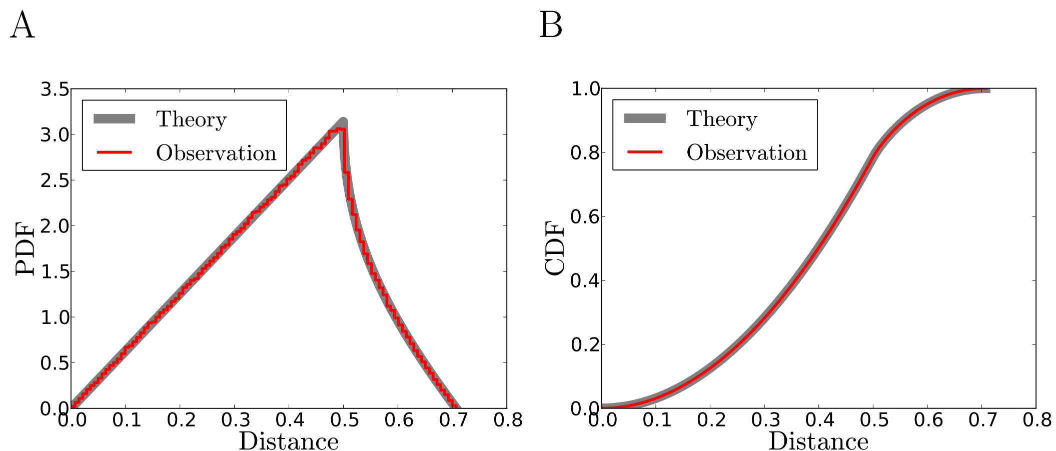


Figure 4.4: PDF (A) and CDF (B) of source-target distances, with layer size $L^2 = 1$, using a constant kernel. The grey lines show the theoretical predictions, and the red lines show the empirical observations. The empirical PDF is plotted with 100 bins.

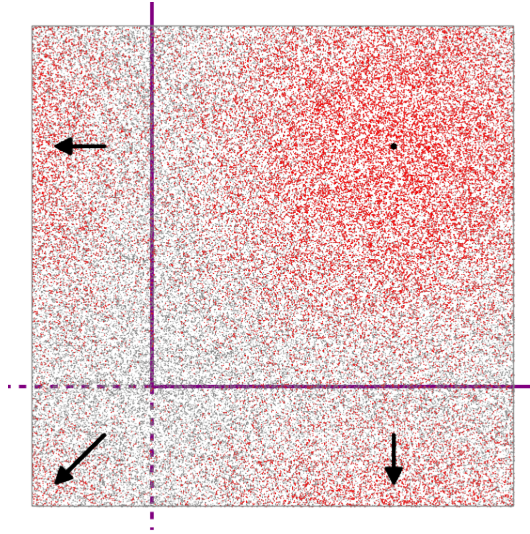


Figure 4.5: Connectivity pattern with periodic boundary conditions and with the source node located at $(L/4, L/4)$. The purple line indicates the mask, and the arrows indicate the direction of shortest distance to the source node. The distribution of distances from the source node to connected nodes is the same as if the source node was located in the center.

tial, and Gaussian kernel, respectively. The p-value for the exponential kernel, $p = 0.004$, is suspiciously low. It appears, however, to be a statistical fluke. Rerunning the test 100 times with different master seed values results in p-values seemingly uniformly distributed on (0, 1). A KS test of uniformity returns a p-values of 0.365.

The tests can also be run with NEST using multiple VPs. This can be done by instantiating the test class with an extra argument `threads`. With 4 threads, there is no evidence of deviations from the expected distributions, both with constant ($p = 0.321$), linear ($p = 0.977$), exponential ($p = 0.614$), and Gaussian ($p = 0.899$) kernel.

Sensitivity

As before, a control algorithm is implemented. It performs a Bernoulli trial on each node, with a probability $p = \mathcal{P}(D)$, given by the kernel, of success (a connection being made). By supplying the `ks_test` method (or the `z_test` method) with an extra argument `control=True`, NEST's connection algorithm is swapped with this control algorithm.

The control algorithm is used to examine the sensitivity of the test to errors and biases deliberately introduced into the data, as well as what effect different parameters have on the sensitivity. Note again that the reported p-values are only examples; a re-run with a different PRNG seed value will give a different p-value.

One bias was introduced by adding a constant $c = 0.01$ to the Gaussian

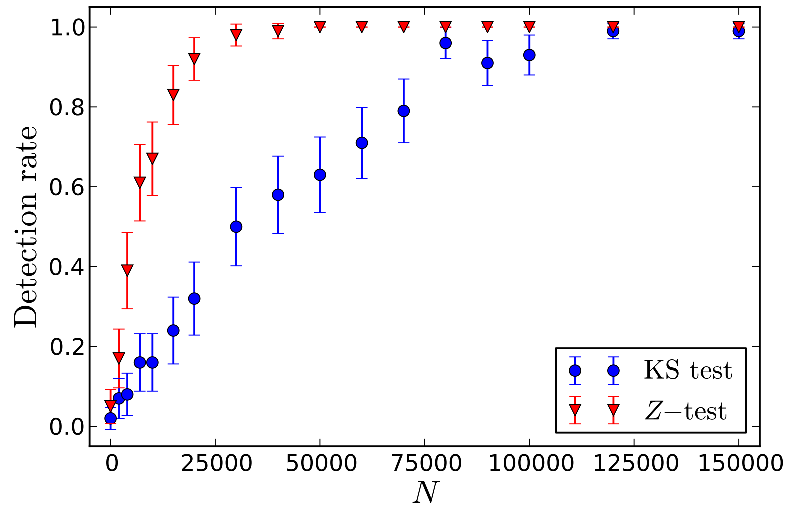


Figure 4.6: Detection rate, e.g., proportion of tests that detect a problem with a bias deliberately introduced into the algorithm, as a function of N, for the KS test (blue) and the Z-test (red). Error bars show 90% confidence intervals. For this particular bias, the Z-test has a higher detection rate than the KS test.

kernel. This bias was easily detected with $N = 1{,}000{,}000$ ($p = 2.50 \times 10^{-29}$), as well as with $N = 100{,}000$ ($p = 6.43 \times 10^{-5}$). With $N = 10{,}000$, the bias was detected at significance level $\alpha = 0.05$ ($p = 0.0159$).

A second bias was introduced by increasing the distances passed to the Gaussian kernel by 1%. This was detected with $N = 1{,}000{,}000$ ($p = 1.59 \times 10^{-5}$), as well as with $N = 100{,}000$ ($p = 8.23 \times 10^{-3}$), but not with $N = 10{,}000$ ($p = 0.585$).

A third bias was introduced by excluding 1% of the nodes, randomly chosen, as potential targets. This bias is not detected by the KS test with any N, because there is no change in the overall distribution of connections. The Z-test, however, detects the bias, both with $N = 1{,}000{,}000$ ($p = 6.66 \times 10^{-16}$) and with $N = 100{,}000$ ($p = 5.32 \times 10^{-4}$), though not with $N = 10{,}000$ ($p = 0.988$).

As noted earlier, the p-values reported above are only examples. To get a sense for how consistently biases are detected, and how the sensitivity varies with N, we again introduce the first bias into the algorithm, i.e., we add a constant $c = 0.01$ to the Gaussian kernel. We then choose the level of significance $\alpha = 0.05$, and consider p-values below α to be a detection. Running the tests 100 times, each time with a different seed, the detection rate, i.e., proportion of times the bias is detected, can be found. In Figure 4.6, this detection rate is plotted as a function of the number of nodes N. Both tests consistently and reliably detect the bias for large N, though the sensitivity falls off rapidly with decreasing N. It is worth noting that the Z-test is more sensitive to this particular bias. Other biases, though, that change the distribution of distances, without affecting the overall connection count, are best detected by the KS

test.

Generally, both tests are quite sensitive to a range of different errors and biases, as long as N is large ($\gtrsim$ 100,000). Biases that do not change the distribution, but do affect the total number of connections, are not detected by the KS test, but such biases are detected quite consistently by the Z-test, as long as N is large.

4.2 Three-dimensional space

The procedure for testing the connectivity pattern of spatially structured networks in three-dimensional space is very similar to that for two-dimensional space, but the expressions for the density of potential targets at a given distance are different.

Let one node be centered on a cubic $L \times L \times L$ layer. It can connect to N other nodes, uniformly distributed on the three-dimensional layer, with a cubic $L \times L \times L$ mask, and a connection probability determined by a kernel $\mathcal{P}(D)$. The distance from node i to node j is

$$D_{ij} = \sqrt{\sum_{m=1}^{k} \Delta x_{ij}^2}, \tag{4.13}$$

where k now is three. With periodic boundary conditions,

$$\Delta x_{ij} = \begin{cases} |(x_i)_m - (x_j)_m| & \text{for } |(x_i)_m - (x_j)_m| \leq L/2 \\ L - |(x_i)_m - (x_j)_m| & \text{for } |(x_i)_m - (x_j)_m| > L/2 \end{cases} \tag{4.14}$$

The average node density is $\rho_0 = N/L^3$. The radial distribution function (RDF) $\rho(D)$ is proportional to the surface area of a sphere with radius D. Thus, at a distance $D \in [0, L/2]$, the RDF is $\rho(D) = \rho_0 4\pi D^2$. For $D \in (L/2, L/\sqrt{2}]$, part of the sphere is outside the cubic mask. Specifically, a spherical cap will stick out of each of the six sides of the cube, as seen in Figure 4.7B. The surface area of each spherical cap is $2\pi Dh$, where $h = D - L/2$ is the height of the cap. Subtracting the surface area of these caps from the total surface area of the sphere, we get $4\pi D^2 - 6\,(2\pi Dh) = 2\pi D\,(3L - 4D)$, and the RDF becomes $\rho(D) = \rho_0 2\pi D\,(3L - 4D)$. For $D \in (L/\sqrt{2}, L\sqrt{3}/2]$, the derivation becomes somewhat involved. The six spherical caps have "grown" to overlap each other near each of the 12 edges of the cube. This situation is seen in Figure 4.7C. If we subtract the surface area C of the spherical caps from the total area A, this overlap E has been subtracted twice, and has to be added again. Calling the surface area of the sphere inside the cube I, we can write

$$I = A - 6C + 12E, \tag{4.15}$$

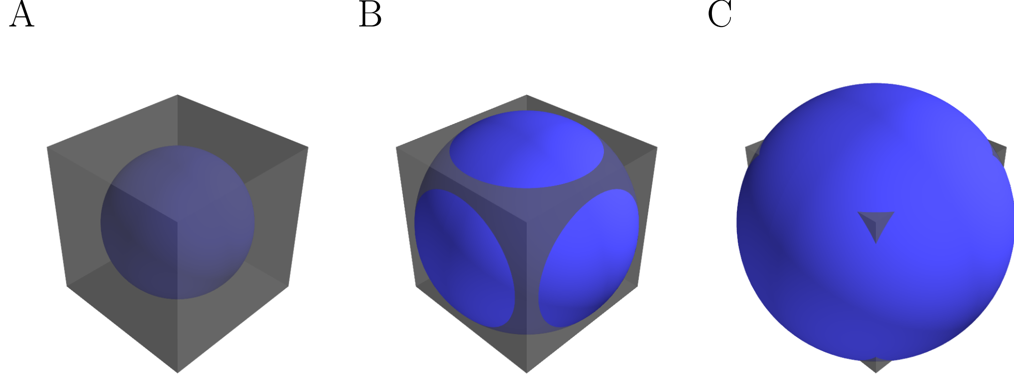


Figure 4.7: A: For distances $D \in [0, L/2]$, the entire sphere is inside the cubic mask. B: For $D \in (L/2, L/\sqrt{2}]$, only part of the sphere is inside the mask, while six spherical caps are outside. C: For $D \in (L/\sqrt{2}, L\sqrt{3}/2]$, the six caps have "grown" to overlap each other.

where $A = 4\pi D^2$ and $C = 2\pi D\,(D - L/2)$. To find E, consider Figure 4.8. The circle outlines one of the spherical caps, sitting on top of one of the cube's sides. The sphere intersects one of the cube's edges at points P and Q. A great circle passing through P and Q bisects the area E into two equal halves. Two more great circles pass through the center point R on the cap's surface and P and Q, respectively. The spherical triangle PQR is further bisected into two equal spherical triangles by a fourth great circle passing through R. One such half (blue triangle in the figure) has a surface area $T = D^2\,(\alpha + \beta + \gamma - \pi)$, where α, β, and γ are the corner angles. The area $\frac{1}{2}E$ can now be found by subtracting $2T$ from the fraction $2\gamma/2\pi$ of the spherical cap's surface area C which is bounded by the two great circle arcs RP and RQ, i.e., $E = 2\,((2\gamma/2\pi)C - 2T)$. The area I of the sphere inside the cube is therefore

$$
\begin{aligned}
I &= A - 6C + 12E & (4.16)\\
&= A - 6C + 12\left(2\left(\tfrac{2\gamma}{2\pi}C - 2T\right)\right) & (4.17)\\
&= A - 6C\left(1 - \tfrac{\gamma}{\pi}\right) - 48T. & (4.18)
\end{aligned}
$$

Summarizing, the RDF is

$$
\rho(D) = \begin{cases} \rho_0 A & \text{for } 0 \le D \le \frac{L}{2} \\ \rho_0\,(A - 6C) & \text{for } \frac{L}{2} < D \le \frac{L}{\sqrt{2}} \\ \rho_0\left(A - 6C\left(1 - \frac{\gamma}{\pi}\right) - 48T\right) & \text{for } \frac{L}{\sqrt{2}} < D \le \frac{L\sqrt{3}}{2} \\ 0 & \text{otherwise,} \end{cases} \qquad (4.19)
$$

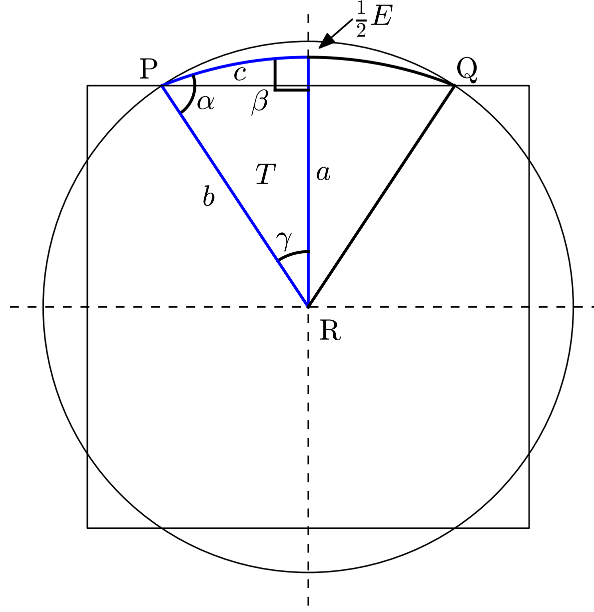


Figure 4.8: Illustration of the derivation of the surface area E of the intersection between adjacent spherical caps. The square is one of the cube's sides, and the circle outlines one of the spherical caps, a portion of the sphere bounded by the extended plane of the cube's side. The area E is seen on the top of the figure, bisected by a great circle passing through points P and Q. The lines RP and RQ are also arcs of great circles. The surface area PQR is $2T$. The area $\frac{1}{2}E$ can be found by subtracting $2T$ from the fraction $\frac{2\gamma}{2\pi}$ of the surface area of the spherical cap bounded by RP and RQ.

where

$$A = 4\pi D^2 \tag{4.20}$$

$$C = 2\pi D \left(D - \tfrac{L}{2}\right) \tag{4.21}$$

$$T = D^2 (\alpha + \beta + \gamma - \pi) \tag{4.22}$$

$$\alpha = \sin^{-1}\left(1/\sqrt{2 - \tfrac{L^2}{2D^2}}\right) \tag{4.23}$$

$$\beta = \tfrac{\pi}{2} \tag{4.24}$$

$$\gamma = \sin^{-1}\left(\sqrt{(1 - \tfrac{L^2}{2D^2})/(1 - \tfrac{L^2}{4D^2})}\right). \tag{4.25}$$

As before, the PDF $f(D)$ of source-target distances is the normalized product of the RDF $\rho(D)$ and some kernel $\mathcal{P}(D)$, and the CDF $F(D)$ can be found by numerical integration of $f(D)$. Figure 4.9 shows an exemplary PDF and CDF, together with the corresponding connectivity pattern.

The KS test is used to compare this theoretical CDF with the observed EDF. A Z-test that compares the observed number of connections with the expected number is also implemented. It works the same way as the Z-test implemented for 2D layers.

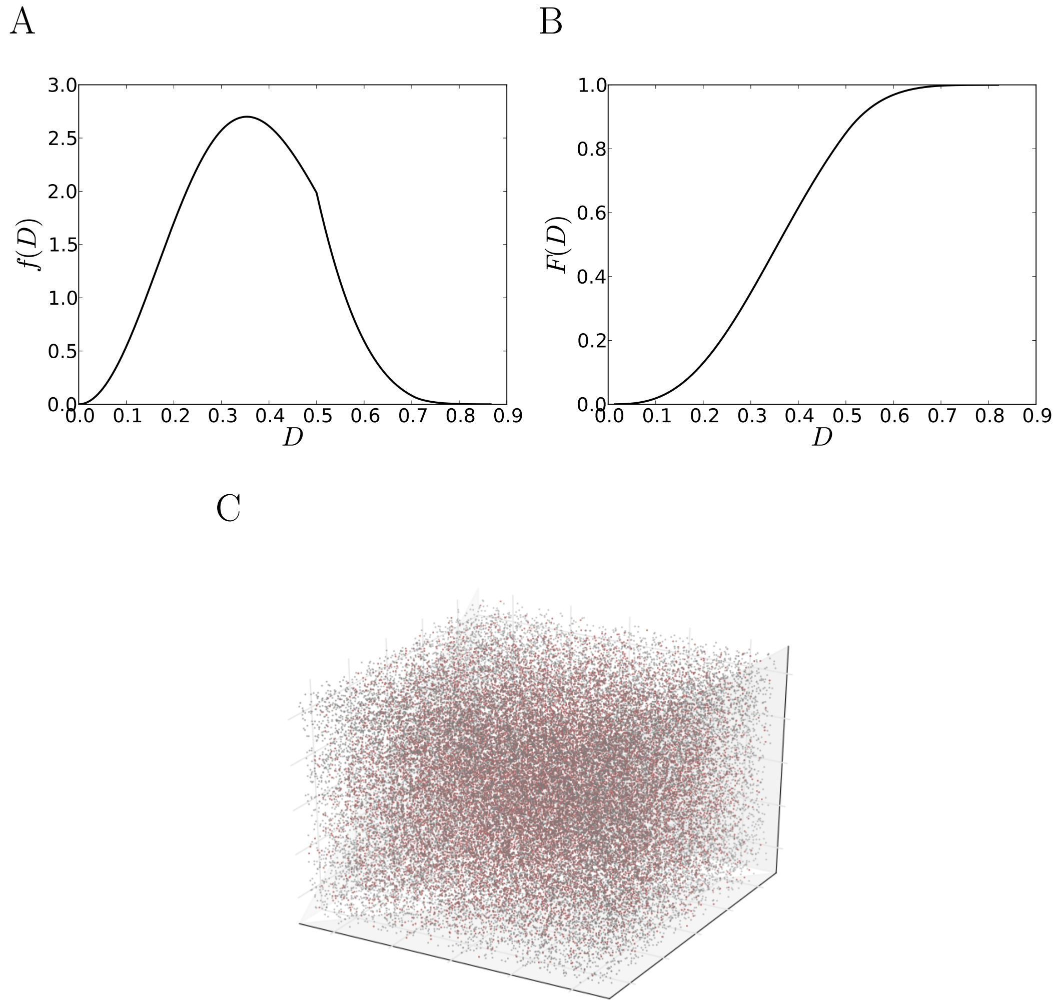


Figure 4.9: Exemplary PDF (A), CDF (B) and connectivity pattern (C) for a spatially structured network in 3D space. A Gaussian kernel is used, and the layer and mask size is $L^3 = 1$.

4.2.1 Implementation

A script implementing the test procedure can be found in Appendix F. The usage is similar to that of 2D layers, and is demonstrated in the main section of the module. Unless `kernel_params` is specified, suitable defaults are used.

4.2.2 Results

Using the script in Appendix F, NEST's connection algorithm for 3D spatially structured networks can be tested. With $L = 1$, $N = 1{,}000{,}000$, there appears to be close agreement between the expected and the observed distribution of

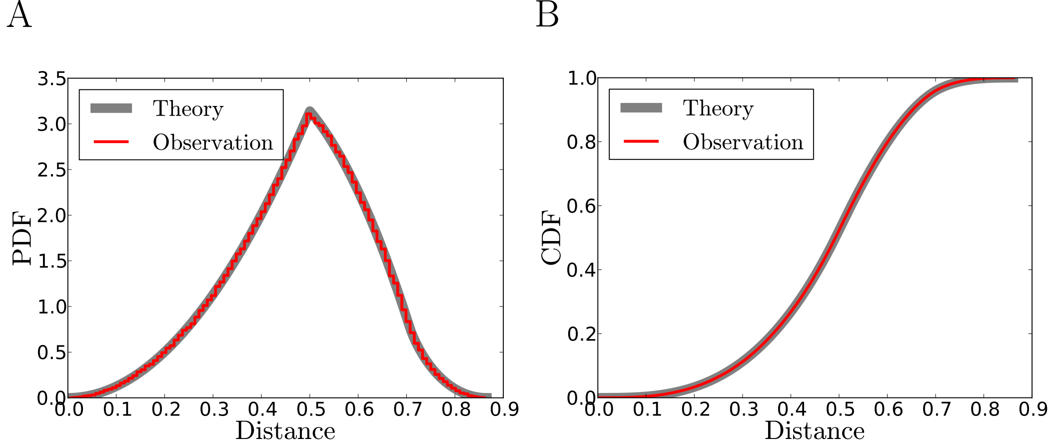


Figure 4.10: PDF (A) and CDF (B) of source-target distances, with layer size $L^3 = 1$, using the constant kernel. The grey lines show the theoretical predictions, and the red lines show the empirical observations. The empirical PDF is plotted with 100 bins.

source-target distances for all four kernels (constant: $p = 0.726$, linear: $p = 0.978$, exponential: $p = 0.803$, and Gaussian: $p = 0.770$). The theoretical and observed PDF and CDF for constant kernel are shown in Figure 4.10. When the source node is moved to $(L/4, L/4, L/4)$ so that the periodic boundary conditions come into play, there is still a close agreement between theory and observation (constant: $p = 0.965$, linear: $p = 0.367$, exponential: $p = 0.217$, and Gaussian: $p = 0.707$), and the same holds when the number of VPs is increased to 4 (constant: $p = 0.726$, linear: $p = 0.600$, exponential: $p = 0.852$, and Gaussian: $p = 0.561$).

Sensitivity

The same control algorithm that was implemented for 2D layers is implemented for 3D layers. It can be used both with the KS test and with the Z-test. We use it again to test the sensitivity of the test procedure to the same biases as for 2D layers. As before, we note that the reported p-values are only examples.

The first bias, introduced by having the Gaussian kernel function return a too high value, with a constant $c = 0.01$ added, was detected by the KS test with $N = 1{,}000{,}000$ ($p = 1.52 \times 10^{-56}$) and $N = 100{,}000$ ($p = 1.48 \times 10^{-6}$), but not with $N = 10{,}000$ ($p = 0.350$). It was detected by the Z-test with $N = 10{,}000$ ($p = 6.25 \times 10^{-5}$) as well as with larger N, but not with $N = 1{,}000$.

The second bias, where the distances passed to the Gaussian kernel function are too high by 1%, is detected by the KS test with $N = 1{,}000{,}000$ ($p = 1.40 \times 10^{-9}$) and $N = 100{,}000$ (at significance level $\alpha = 0.05$; $p = 0.019$), but not with $N = 10{,}000$. The Z-test also detects the bias with $N = 100{,}000$ ($p = 5.46 \times 10^{-8}$), but not with $N = 10{,}000$ ($p = 0.818$).

The third bias, where a randomly selected 1% of the nodes are excluded as potential targets, can not be detected by the KS test, regardless of N. The Z-test, however, detects the bias with $N = 1{,}000{,}000$ ($p = 3.38 \times 10^{-10}$), $N = 100{,}000$ ($p = 1.05 \times 10^{-4}$), and $N = 10{,}000$ (at $\alpha = 0.05$; $p = 0.035$), but not with $N = 1{,}000$ ($p = 0.388$).

Overall, the KS test is sensitive when N is large, but the sensitivity falls of rapidly with decreasing N. Perhaps surprisingly, the Z-test is often more sensitive than the KS test, depending on the nature of the bias we wish to detect. The two tests are in some ways complementary to each other. The KS test detects biases that changes the overall distribution of source-target distances, but it does not detect biases that affect the total number of connections without changing the distribution. The Z-test does detects biases that affect the total number of connections without changing the distribution.

4.3 Automated test procedure

We will now describe the automated test procedure for spatially structured networks, both in 2D and 3D space.

To maintain a high sensitivity while reducing the rate of false positives and the execution time, an adaptive testing strategy is used. First, a single KS test is run. If the resulting p-value is suspiciously low, a two-level test is run, comparing the output of n_{runs} KS tests to the expected uniform distribution, thereby either confirming or allaying our suspicion.

As discussed earlier, certain errors can not be detected by the KS test, but are detected by the Z-test. We therefore include the Z-test in the automated test procedure. The adaptive strategy can be used for the Z-test as well as for the KS test, re-running the Z-test n_{runs} times if the first test results in a suspicious p-value.

Let α_1 be the value below which the p-value from the single KS test (or Z-test) is considered suspicious, and α_2 be the value below which the p-value from the two-level test is considered too extreme, resulting in the rejection of H_0. The fraction of false positives will be $\alpha = \alpha_1\alpha_2$. If k tests are included in the test suite, the number of false positives is binomially distributed. The probability of seeing one or more false positives is therefore

$$\begin{aligned}
\text{Prob}(x \geq 1) &= 1 - \text{Prob}(x = 0) \\
&= 1 - \binom{k}{0}(\alpha_1\alpha_2)^0(1 - \alpha_1\alpha_2)^k \\
&= 1 - (1 - \alpha_1\alpha_2)^k
\end{aligned}$$

The choice of α_1 and α_2 might thus depend on k, as well as other factors, such as the acceptable rate of false positives and the desired sensitivity and

the tests. The number of nodes N in the network and number of re-runs n_{runs} upon suspicion must also be chosen. Large values will clearly increase sensitivity, but the available computation time is a limiting factor.

4.3.1 Implementation

An implementation of the automated test procedure is found in Appendix G. Each of the four kernels (constant, linear, exponential, and Gaussian) are tested. In addition, tests are done with the source node shifted away from the center, and with multiple VPs. These six test configurations are tested both with the KS test and the Z-test, and both in two-dimensional and three-dimensional space. A total of 24 tests is thus run. With $\alpha_1 = \alpha_2 = 0.01$, the entire test suite will falsely report a problem with a probability $1 - (1 - 0.01 \times 0.01)^{24} = 2.40 \times 10^{-3}$. With $N = 100{,}000$ nodes in the network and $n_{\text{runs}} = 100$ re-runs upon suspicion, the test suite will in most cases complete in a couple of minutes.

Chapter 5

Discussion

We have developed tests for two main types of connection patterns, namely, random convergent and divergent connections with multapses and autapses allowed, and structured networks in two- and three-dimensional space with distance-dependent connection probability. Both tests have been implemented as Python test suites and have been used to test the connection routines in NEST. We emphasize that the test suites can be adapted to work with other simulators, simply by changing the function calls for generating the network and retrieving the resulting connections.

For random connections, a two-level test was proposed. Pearson's chi-squared test is used to test whether nodes are selected randomly and with equal probability. The resulting p-values are then compared with the expected uniform distribution using the Kolmogorov-Smirnov (KS) test. Advantages of this approach are increased sensitivity and the ability to detect too good fits as well as poor fits.

For spatially structured networks, expressions for the radial distribution function were derived, both for two- and three-dimensional space. The observed distribution of source-target distances could then be compared with the expected distribution, found as the normalized product of the radial distribution function $\rho(D)$ and the distance-dependent connection probability (kernel) $\mathcal{P}(D)$, using the KS test.

We demonstrated the utility of the tests, both that they are able to detect a range of errors, and that they do not fail more often than expected when there is no error in the algorithms tested. To actually show this, rather than assume it is so, is important, because assumptions used when developing the tests might not be as accurate as assumed. For example, in the case of Pearson's chi-squared goodness-of-fit test, the assumption of uniformly distributed p-values is not strictly true, due to the discreteness of the p-values. And indeed, for very sparse data, the two-level test procedure used for testing random convergent or divergent connections was shown to not be reliable. It was demonstrated, however, that both tests (1) detect many deliberately introduced errors, and

(2) do not report more than the expected fraction of false positives under H_0 (as long as data is not too sparse).

The tests were used on NEST's probabilistic connection algorithms, with a range of different parameters, under different conditions. No evidence of errors or biases was found. For example, three-dimensional spatially structured networks with 1,000,000 nodes were created using each of the four kernels (constant, linear, exponential, and Gaussian), and the distribution of distances between connected nodes was tested using the KS test, resulting in the p-values 0.726, 0.978, 0.803, and 0.770, consistent with our expectations. This of course does not guarantee that no bias exists. Very small biases, or biases of a kind not easily detectable by the tests (e.g., biases caused by patterns in the underlying PRNG), might, indeed probably do, exist. Still, our confidence in the simulated connection patterns, and therefore the scientific findings based them, has grown.

An additional goal of this work was to develop automated test suites. An adaptive test strategy was proposed as a solution to the extra challenges this entails. A single test is first done on the algorithm being tested. If the resulting p-value is deemed suspicious, a two-level test is performed, comparing the p-values from tests of multiple network realizations with the expected uniform distribution. Using this strategy, the automated test suites achieve a low rate of false positives, fast run time, and a fairly high sensitivity. In certain cases one might want to opt for a safer alternative and do a small number of initial tests, instead of one, to determine whether more tests should be run. This will increase the sensitivity, while run time and rate of false positives will increase.

Perspectives for future research

Variants of the probabilistic network types tested in this work exist, and tests have yet to be developed for these. The test for random convergent or divergent connection routines developed here, for example, assumes that autapses and multapses are allowed, and cannot be used when these are disallowed. Networks with disallowed autapses are relatively straightforward to test. We can simply make sure no node is connected to itself, and do a chi-squared test with the number of nodes available reduced by one. For disallowed multapses, however, an altogether different distribution will result.

For networks with spatial structure, tests for networks with a prescribed in- or out-degree C, as well as a kernel, remain to be developed. It is not entirely clear what is expected from such a connection routine, as a conflict between the two rules occur. In some cases, the value of the kernel is in this case interpreted not as a connection probability, but instead as relative probabilities (Plesser and Austvoll 2009). With kernel value for node i equal to k_i, the relative probability of connecting to node i is $p_i = k_i/K$, where $K = \sum_i k_i$. For such a connection routine, one possible test strategy is to use the chi-squared test,

with p_iC as the expectations.

Another case worth testing is spatially structured networks with open boundary conditions. When the source node is arbitrarily positioned, the boundary effects will in this case lead to somewhat involved expressions for the radial distribution of nodes.

Appendix A

NEST - A short tutorial

NEST (NEural Simulation Tool, Gewaltig and Diesmann (2007)) is a simulation environment developed by the NEST Initiative, capable of running simulations of large networks of point neurons. NEST is open source and can be downloaded free of charge from nest-initiative.org. A brief introduction is given here. This section is based on the NEST tutorial by Gewaltig, Morrison, and Plesser (2012) and the NEST Topology User Manual by Plesser and Enger (2012). It mainly describes the functions implementing the connection algorithms tested in this thesis. A Python interface, PyNEST (Eppler et al. 2009), is available, and will be used here. Version 2.2.0 of NEST is used. To import the PyNEST module into Python we use the following command.

```
import nest
```

The concepts of *nodes* and *connections* described earlier are used in NEST. **Nodes** can be neurons, devices, or sub-networks. The neurons are either point-neurons (neurons with a single compartment) or neurons with a small number of compartments. They can be based on one of the many built-in neuron models, some of which are listed in Table A.1. The default parameters of each of these models can be modified. Due to NEST's modular architecture, researchers can also create their own models from scratch.

Devices are nodes used to either stimulate or measure the activity of neurons. Some of the available devices are listed in Table A.2. Sub-networks (or subnets) are nodes who themselves are comprised of multiple nodes. Subnets can also be created inside other subnets. Using nested subnets we can create complex hierarchical structures of neurons with as many levels as we want.

The Python code below will create a neuron of type "iaf_neuron", using the default parameters for the model.

```
neuron1 = nest.Create('iaf_neuron')
```

Every node created is assigned number, called a global identifier (GID). This number is returned and assigned to the variable called `neuron1`. To create a

Name	Description
aeif_cond_alpha	Conductance based exponential integrate-and-fire neuron model according to Brette and Gerstner (2005).
ginzburg_neuron	Binary stochastic neuron with sigmoidal activation function.
hh_psc_alpha	Spiking neuron using the Hodkin-Huxley formalism.
ht_neuron	Neuron model after Hill and Tononi (2005).
iaf_cond_alpha	Simple conductance based leaky integrate-and-fire neuron model. Post-synaptic change of conductance modelled by an alpha function.
iaf_cond_exp	Simple conductance based leaky integrate-and-fire neuron model. Post-synaptic change of conductance modeled by an exponential function.
iaf_neuron	Leaky integrate-and-fire model with alpha-function shaped synaptic currents.
izhikevich	Implementation of the simple spiking neuron model introduced by Izhikevich (2003).
mat2_psc_exp	Non-resetting leaky integrate-and-fire neuron model with exponential PSCs and adaptive threshold.
parrot_neuron	Neuron that repeats incoming spikes.
pp_psc_delta	Point process neuron with leaky integration of delta-shaped PSCs.
sli_neuron	The sli_neuron is a model whose state, update and calibration can be defined in SLI.

Table A.1: A few of the neuron models available in NEST.

neuron with some non-default parameters, say, the threshold potential V_{th} and the reset potential V_{reset}, the code will look as follows.

```
neuron2 = nest.Create('iaf_neuron',
                      params={'V_th': -50.0, 'V_reset': -65.0})
```

To create many identical nodes, the desired number of nodes is passed to `Create` as the second argument. In the following code 1,000 neurons are created, and the GIDs for all the neurons are stored as a list named `neurons`.

```
neurons = nest.Create('iaf_neuron', 1000)
```

Connections in NEST can represent synapses between neurons, but can also be connections from neurons to other types of nodes (devices and subnets). All connections are *directed*, *weighted* and *delayed*. Directed means information

Name	Description
ac_generator	This device produce an ac-current.
correlation_detector	Device for evaluating cross correlation between two spike sources
dc_generator	The DC-Generator provides a constant DC input to the connected node.
gamma_sup_generator	Simulate the superimposed spike train of a population of gamma process.
multimeter	Device to record analog data from neurons.
noise_generator	Device to generate Gaussian white noise current.
poisson_generator	Simulate neuron firing with Poisson processes statistics.
spike_detector	Device for detecting single spikes.
spike_generator	A device which generates spikes from an array with spike-times.
voltmeter	Device to record membrane potentials from neurons.

Table A.2: Some of the devices available in NEST.

travels in one direction. Weighted means the strength of connections can be varied. Delayed means it takes time for information to travel from one node to a connected node.

To manually connect one pair of nodes using the default synapse model "static_synapse" with the default weight of 1.0 and the default delay of 1.0 milliseconds we can use the `Connect` function:

```
nest.Connect(neuron1, neuron2)
```

To use a weight of −1.5 (negative weights means the connection will be inhibitory), a delay of 0.5 ms, and the synapse model "tsodyks_synapse", we pass some extra arguments to `Connect` as follows.

```
nest.Connect(n1, n2, -1.5, 0.5, model='tsodyks_synapse')
```

Table A.3 lists some of the synapse models in NEST. Several synapse models with short-term and long-term plasticity are available.

`Connect` can also be used to create many one-to-one connections between two lists of nodes. However, if we wish to make connections that are more complex than simple one-to-one connections, such as convergent and divergent connections, several other functions are available. One example is `RandomConvergentConnect`. The code snippet below first defines three variables N_{s}, N_{t} and C. It then creates N_{s} neurons, which will serve

Name	Description
cont_delay_synapse	Synapse type for continuous delays.
ht_synapse	Synapse with depression after Hill and Tononi (2005).
static_synapse	Synapse type for static connections.
stdp_dopamine_synapse	Synapse type for dopamine-modulated spike-timing dependent plasticity.
stdp_synapse	Synapse type for spike-timing dependent plasticity.
tsodyks_synapse	Synapse type with short term plasticity.

Table A.3: Some of the synapse models available in NEST.

as *source* neurons, and N_t *target* neurons. It then connects them using `RandomConvergentConnect`.

```
N_s = 10
N_t = 10
C = 3
source_neurons = nest.Create('iaf_neuron', N_s)
target_neurons = nest.Create('iaf_neuron', N_t)
nest.RandomConvergentConnect(source_neurons, target_neurons, C)
```

The function call

```
nest.GetConnections(source_neurons)
```

will return the resulting connections. `RandomConvergentConnect` considers each node in the second list (`target_neurons`) in turn, and connects C randomly chosen nodes from the first list (`source_neurons`). Hence, the target nodes will all have an in-degree (number of incoming connections) equal to C, whereas the source nodes might have different out-degrees. Note that two or more connections (multapses) can be drawn between the same pair of nodes.

A similar function, `RandomDivergentConnect`, works in very much the same way, but it iterates over the *source* nodes instead of the target nodes, and draws random nodes from the *targets* instead of the sources. As a result, the source nodes will now all have the same out-degree C, while the in-degrees of the target nodes will vary. An example of a random divergent network was shown in Figure 1.1.

After having created all nodes and connected them, the simulation can be started with the `Simulate` function. `Simulate` only takes one argument; the number of milliseconds to simulate. The following example, borrowed from Gewaltig, Morrison, and Plesser (2012), is a simple simulation script. A neuron

receives stimuli (events) from one AC generator and two poisson generators. A voltmeter is also connected to the neuron to record the membrane potential. The simulation is run for 1,000 ms, and a plot of the membrane potential as a function of time is displayed.

```
import nest
import nest.voltage_trace
neuron = nest.Create('iaf_neuron')
sine = nest.Create('ac_generator', 1,
                   {'amplitude': 100.0, 'frequency': 2.0})
noise = nest.Create('poisson_generator', 2,
                    [{'rate': 70000.0}, {'rate': 20000.0}])
voltmeter = nest.Create('voltmeter', 1, {'withgid': True})
nest.Connect(sine, neuron)
nest.Connect(voltmeter, neuron)
nest.ConvergentConnect(noise, neuron, [1.0, -1.0], 1.0)
nest.Simulate(1000.0)
nest.voltage_trace.from_device(voltmeter)
```

Spatially structured networks and the NEST Topology module

The simulation of spatially structured networks is increasingly a popular tool. A spatially structured network is a network where connection probabilities and -properties are determined by the spatial position of the nodes. With NEST, such networks can be created easily using the NEST Topology module (Plesser and Austvoll 2009; Plesser and Enger 2012).

To use NEST Topology, the module has to be imported as show below. For convenience it is given the shorthand name “topo”.

```
import nest
import nest.topology as topo
```

In Topology, nodes are placed on *layers*. Layers are similar to subnets, but with extra information about the spatial position of each node. The term “layer” often refers to a two-dimensional structure, but layers in NEST Topology can be both two- and three-dimensional. They are either grid-based or free. In *grid-based layers*, nodes are placed on a Cartesian grid, while on *free layers*, nodes can be placed at arbitrary locations. The code below will create a simple two-dimensional grid-based layer.

```
layer_specs = {'elements': 'iaf_neuron',
               'rows': 10, 'columns': 10, 'extent': [2.0, 2.0]}
layer = topo.CreateLayer(layer_specs)
```

First a dictionary, `layer_specs`, is created, containing specifications for the layer. The node type is set to "iaf_neuron". The `rows` and `columns` entries specify the number of desired rows and columns of neurons, and `extent` is the size of the layer. Additional parameters can be included, such as the x and y position of the center of the layer (default is (0, 0)). The layer is assigned a GID, which is returned by `CreateLayer` and assigned to the variable `layer`.

To create a *free* layer, the position of every node must be passed to `CreateLayer`. If a list of positions is given, a node will be created at each. In the example below, three nodes are positioned on a two-dimensional layer.

```
pos = [[-0.3, 0.3], [0.4, 0.2], [0.0, -0.4]]
layer = topo.CreateLayer({'elements': 'iaf_neuron',
                          'positions': pos})
```

Here the extent is not specified, so the layer will have the default extent of 1.0×1.0.

To place a larger number of neurons at pseudorandom locations, we have to import and use a pseudorandom number generator (PRNG), such as the one supplied with `numpy`. Here, 1,000 neurons are uniformly distributed over the layer:

```
import numpy as np
x = np.random.uniform(-0.5, 0.5, 1000)
y = np.random.uniform(-0.5, 0.5, 1000)
pos = zip(x, y)
l = topo.CreateLayer({'elements': 'iaf_neuron',
                      'positions': pos})
topo.PlotLayer(l)
```

The last line will display a plot of all the nodes on the layer, similar to the one in Figure A.1. To create a three-dimensional layer we simply add a third component to the node positions.

```
import numpy as np
x = np.random.uniform(-0.5, 0.5, 1000)
y = np.random.uniform(-0.5, 0.5, 1000)
z = np.random.uniform(-0.5, 0.5, 1000)
pos = zip(x, y, z)
l = topo.CreateLayer({'elements': 'iaf_neuron',
                      'positions': pos})
topo.PlotLayer(l)
```

The result is seen in Figure A.2.

For small layers, a large fraction of the nodes will be close to the edge and therefore have fewer neighboring nodes. This might have an undesired effect on the simulation. To emulate the effect of a larger layer, the layers can

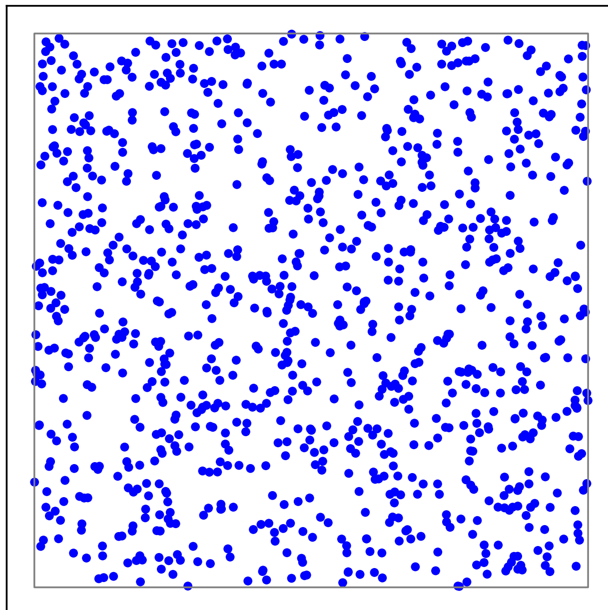


Figure A.1: A free layer with 1,000 nodes. x and y positions are drawn pseudorandomly from $\mathcal{U}(-0.5, 0.5)$.

therefore be given periodic boundary conditions. The layer is effectively bent into a torus, so the left and right side are joined, and the top and bottom is joined.

To create a layer with periodic boundary conditions, an extra entry, `'edge_wrap': True`, is included in the dictionary passed to `CreateLayer`:

```
layer = topo.CreateLayer({'elements': 'iaf_neuron',
                          'positions': pos,
                          'edge_wrap': True})
```

The `ConnectLayers` function is used to connect two layers. One layer is considered the *source* layer, and the other the *target* layer. Connections between layers can be convergent or divergent. When creating a *convergent* connection, each node in the *target* layer is iterated over, and connections are drawn from the *source* layer. When creating a *divergent* connection, each node in the *source* layer is iterated over, and connections are drawn from the *target* layer. The layer which is iterated over is called the *driver*, and the layer from which nodes are drawn is called the *pool* layer.

When connecting two layers, we also specify a mask, and usually a kernel. A *mask* is a boundary around the considered node in the driver layer beyond which no nodes from the pool layer are considered. Masks can be rectangular, circular, or annulus shaped. The *kernel* is the distance-dependent connection probability function. Table A.4 lists all the available kernels in Topology. Users can also create their own kernels.

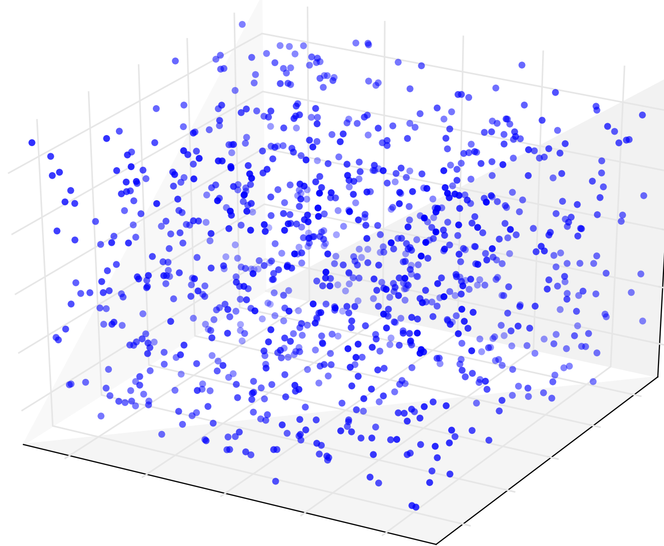


Figure A.2: A free three-dimensional layer with 1,000 nodes, where x, y, and z positions are drawn pseudorandomly from $\mathcal{U}(-0.5, 0.5)$.

In the example below, two identical layers are created, and then connected using a Gaussian kernel. The result can be seen in Figure A.3.

```
1  import nest
2  import nest.topology as topo
3
4  layer_specs = {'elements': 'iaf_neuron',
5                 'rows': 21, 'columns': 21}
6  source_layer = topo.CreateLayer(layer_specs)
7  target_layer = topo.CreateLayer(layer_specs)
8
9  mask = {'rectangular': {'lower_left': [-0.4, -0.4],
10                         'upper_right': [0.4, 0.4]}}
11 kernel = {'gaussian': {'p_center': 1., 'sigma': .2}}
12 conn_specs = {'connection_type': 'convergent',
13               'mask': mask, 'kernel': kernel}
14 topo.ConnectLayers(source_layer, target_layer, conn_specs)
15
16 fig = topo.PlotLayer(target_layer, nodesize = 60,
17                      nodecolor = 'grey')
18
19 center_node = topo.FindCenterElement(source_layer)
20 topo.PlotTargets(center_node, target_layer, fig = fig,
21                  mask = mask, kernel = kernel,
```

Name	Parameters	Function
constant		$p \in [0,1]$
uniform	min, max	$p \in [min, max)$ uniformly
linear	a, c	$p(d) = c + ad$
exponential	a, c, τ	$p(d) = c + ae^{-\frac{d}{\tau}}$
gaussian	$p_{\text{center}}, \sigma, \mu, c$	$p(d) = c + p_{\text{center}} e^{-\frac{(d-\mu)^2}{2\sigma^2}}$
gaussian2D	p_{c}, σ_{x}, σ_{y}, μ_{x}, μ_{y}, ρ, c	$p(d) = c + p_{\text{c}} e^{-\frac{\frac{(d_{\text{x}}-\mu_{\text{x}})^2}{\sigma_{\text{x}}^2} - \frac{(d_{\text{y}}-\mu_{\text{y}})^2}{\sigma_{\text{y}}^2} + 2\rho\frac{(d_{\text{x}}-\mu_x)(d_{\text{y}}-\mu_{\text{y}})}{\sigma_{\text{x}}\sigma_{\text{y}}}}{2(1-\rho^2)}}$

Table A.4: Kernels available in NEST Topology.

```
22                    src_size = 250, src_color = 'red',
23                    tgt_size = 30, tgt_color = 'red',
24                    kernel_color = 'blue')
```

In lines 4 and 5, layer specifications are stored in the variable `layer_specs`. No extent is specified, so the default extent of 1.0×1.0 will be used. In line 6 and 7, these specs are used to create two identical layers, called `source_layer` and `target_layer`. In line 9 through 11, mask and kernel specifications are stored. A rectangular mask is chosen, and the extent of the mask is specified in terms of the locations of the lower left and upper right corners. A Gaussian kernel is chosen, and the two required parameters of the function, the connection probability at the center (`p_center`) and the standard deviation (`sigma`), are set. In line 12 and 13 the connection specifications are set using the already defined mask and kernel, and the connection type set to convergent. In line 14 the layers are actually connected. The rest of the code generates the plot seen in Figure A.3.

It is possible to specify a fixed degree for the driver layer, similar to what is done when using `RandomConvergentConnect` for networks without spatial structure. In this case, each node in the driver layer selects randomly chosen nodes from the pool layer and either connects or not based on the probability given by the kernel, until the number of connections matches the specified degree. This will allow multapses to be formed (unless specifically prohibited). To implement a fixed driver layer degree, only a small change to the parameters given to `ConnectLayers` is needed:

```
conn_specs = {'connection_type': 'convergent',
              'mask': mask, 'kernel': kernel,
              'number_of_connections': 100,
              'allow_multapses': True}
```

Several functions for inspecting different aspects of the network is available, in addition to the plotting functions demonstrated above. All the query

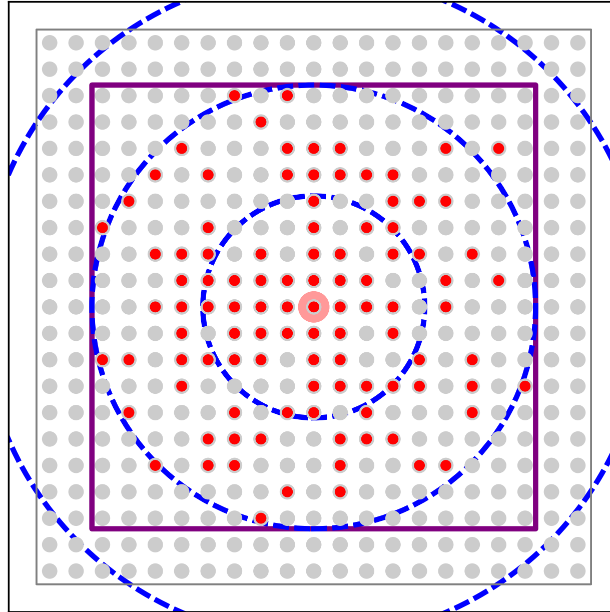


Figure A.3: Plot of a layer, with neurons connected to the source node in the center marked in red. The mask is shown in purple, and the blue rings mark σ, 2σ and 3σ from the Gaussian kernel function. No nodes outside the mask is connected to the source node. As expected, the fraction of connected nodes is higher close to the source node. Note however, that the actual number of connected nodes is higher between the 1σ mark and the 2σ mark than between the center and the 1σ mark. This is because a greater number of nodes are available further away from the center.

functions from the main nest module, such as the `GetTargetNodes` function explained earlier, can also be used on layered networks created with the Topology module. In addition, the topology module provides some extra query functions, such as `FindCenterElement`, `GetTargetPositions`, `Distance`, and `Displacement`. The usage of some of these are demonstrated in the code below, which is a continuation of the previous example.

```
25 targets = topo.GetTargetNodes(center_node, target_layer)[0]
26 distances = topo.Distance(center_node, targets)
27 import matplotlib.pyplot as plt
28 plt.figure()
29 plt.hist(distances, bins=10)
```

This will generate a histogram like the one in Figure A.4, showing the distribution of distances between the center source node and connected nodes in the target layer.

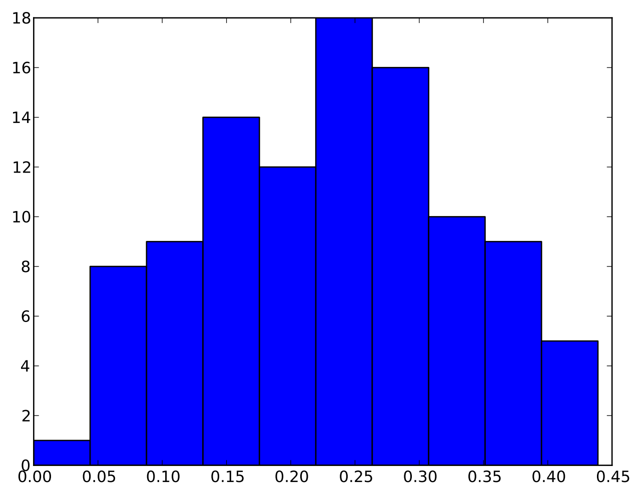


Figure A.4: Distribution of distances between a centered source node and the connected target nodes.

Appendix B

Test script for random convergent connections

```
'''
@author: Daniel Hjertholm

Tests for RandomConvergentConnect.
'''

import numpy
import numpy.random as rnd
import scipy.stats
import matplotlib.pyplot as plt
import nest


class RCC_tester(object):
    '''
    Class used for testing RandomConvergentConnect.
    '''

    def __init__(self, N_s, N_t, C, e_min=10, threads=1):
        '''
        Sets up the experiment, and calculates expected distributions
        for later comparison with the observed distribution.

        Parameters
        ----------
            N_s    : Number of source neurons.
            N_t    : Number of target neurons.
            C      : In-degree (number of connections per
                     target neuron).
            e_min  : Minimum expected number of observations in
                     each bin. Default is 10.
            threads: Set number of local threads. Default is 1.
        '''

        self.N_s = N_s
        self.N_t = N_t
        self.C = C
        self.e_min = e_min
        self.threads = threads

        nest.ResetKernel()
        nest.SetKernelStatus({'local_num_threads': self.threads})
        self.n_vp = nest.GetKernelStatus('total_num_virtual_procs')

        expected_degree = self.N_t * self.C / float(self.N_s)
        if expected_degree < self.e_min:
            raise RuntimeWarning(
                'Expected out-degree (%.2f) is less than e_min (%.2f).' \
                'Increase N_t*C / N_s or decrease e_min.' % \
                (expected_degree, self.e_min))

        self.expected = [expected_degree] * self.N_s

    def _counter(self, x):
        '''
        Count similar elements in list.

        Parameters
        ----------
            x: Any list.

        Return values
        -------------
            list containing counts of similar elements.
        '''

        start = min(x)
        counts = [0] * self.N_s
        for elem in x:
            counts[elem - start] += 1

        return counts

    def _reset(self, msd):
        '''
        Reset the NEST kernel and set seed values.

        Parameters
        ----------
            msd: Master RNG seed.
        '''

        nest.ResetKernel()
        nest.SetKernelStatus({'local_num_threads': self.threads})
        self.n_vp = nest.GetKernelStatus('total_num_virtual_procs')

        # Set PRNG seed values:
        if msd == None:
            msd = rnd.randint(1000000)
```

```
90          msdrange = range(msd, msd + self.n_vp)
91          nest.SetKernelStatus({'grng_seed': msd + self.n_vp,
92                                'rng_seeds': msdrange})
93
94      def _build(self):
95          '''Create all nodes.'''
96
97          self.source_nodes = nest.Create('iaf_neuron', self.N_s)
98          self.target_nodes = nest.Create('iaf_neuron', self.N_t)
99
100     def _connect(self):
101         '''Connect all nodes.'''
102
103         nest.RandomConvergentConnect(self.source_nodes, self.target_nodes,
104                                      self.C, options={'allow_multapses': True})
105
106     def _get_degrees(self, msd):
107         '''
108         Reset NEST, create nodes, connect them, and retrieve the
109         resulting connections.
110
111         Parameters
112         ----------
113             msd: Master RNG seed.
114
115         Return values
116         -------------
117             list containing the out-degrees of the source nodes.
118         '''
119
120         self._reset(msd)
121         self._build()
122         self._connect()
123
124         connections = nest.GetConnections(source=self.source_nodes)
125         source_connections = [conn[0] for conn in connections]
126         degrees = self._counter(source_connections)
127
128         return degrees
129
130     def _get_degrees_control(self, msd):
131         '''
132         Instead of using NEST, this method returns data with the expected
133         multinomial distribution.
134
135         Parameters
136         ----------
137             msd: Master RNG seed.
138
139         Return values
140         -------------
141             list containing the out-degrees of the source nodes.
142         '''
143
144         if msd != None:
145             rnd.seed(msd)
146
147         con = rnd.randint(0, self.N_s, self.N_t * self.C)
148         degrees = self._counter(con)
149
150         return degrees
151
152     def chi_squared_test(self, msd=None, control=False):
153         '''
154         Create a single network and compare the resulting out-degree
155         distribution with the expected distribution using Pearson's chi-squared
156         GOF test.
157
158         Parameters
159         ----------
160             msd    : Master RNG seed.
161             control: Boolean value. If True, _get_degrees_control will
162                      be used instead of _get_degrees.
163
164         Return values
165         -------------
166             chi-squared statistic.
167             p-value from chi-squared test.
168         '''
169
170         if control:
171             degrees = self._get_degrees_control(msd)
172         else:
173             degrees = self._get_degrees(msd)
174
175         # ddof: adjustment to the degrees of freedom. df = k-1-ddof
176         return scipy.stats.chisquare(numpy.array(degrees),
177                                      numpy.array(self.expected), ddof=0)
178
179     def two_level_test(self, n_runs, start_seed=None, control=False,
180                        show_histogram=False, histogram_bins=100,
181                        show_CDF=False):
182         '''
183         Create a network and run chi-squared GOF test n_runs times.
184         Test whether resulting p-values are uniformly distributed
185         on [0, 1] using the Kolmogorov-Smirnov GOF test.
186
187         Parameters
188         ----------
189             n_runs        : number of times to repeat chi-squared test.
190             start_seed    : First PRNG seed value.
191             control       : boolean value. If True, _get_degrees_control will be
192                             used instead of _get_degrees.
193             show_histogram: Specify whether histogram should be displayed.
```

```
194            histogram_bins: Number of histogram bins.
195            show_CDF      : Specify whether EDF should be displayed.
196
197        Return values
198        -------------
199            KS statistic.
200            p-value from KS test.
201        '''
202
203        self.pvalues = []
204
205        if start_seed == None:
206            for i in range(n_runs):
207                print 'Running test %d of %d.' % (i + 1, n_runs)
208                chi, p = self.chi_squared_test(None, control)
209                self.pvalues.append(p)
210        else:
211            seed_jump = self.n_vp + 1
212            end_seed = start_seed + n_runs * seed_jump
213            for seed in range(start_seed, end_seed, seed_jump):
214                print 'Running test %d of %d.' % \
215                    (1 + (seed - start_seed) / seed_jump, n_runs)
216                chi, p = self.chi_squared_test(seed, control)
217                self.pvalues.append(p)
218
219        ks, p = scipy.stats.kstest(self.pvalues, 'uniform',
220                                   alternative='two_sided')
221
222        if show_CDF:
223            plt.figure()
224            self.pvalues.sort()
225            y = [(i + 1.) / len(self.pvalues)
226                 for i in range(len(self.pvalues))]
227            plt.step([0.0] + self.pvalues + [1.0], [0.0] + y + [1.0])
228            plt.xlabel('P-values')
229            plt.ylabel('Empirical distribution function')
230
231        if show_histogram:
232            plt.figure()
233            plt.hist(self.pvalues, bins=histogram_bins)
234            plt.xlabel('P-values')
235            plt.ylabel('Frequency')
236
237        if show_CDF or show_histogram:
238            plt.show(block=True)
239
240        return ks, p
241
242
243 if __name__ == '__main__':
244    test = RCC_tester(N_s=1000, N_t=1000, C=1000)
245    ks, p = test.two_level_test(n_runs=1000, start_seed=0)
246    print 'KS test statistic:', ks
247    print 'p-value of KS-test of uniformity:', p
```

Appendix C

Test script for random divergent connections

```
'''
@author: Daniel Hjertholm

Tests for RandomDivergentConnect.
'''

import numpy
import numpy.random as rnd
import scipy.stats
import matplotlib.pyplot as plt
import nest


class RDC_tester(object):
    '''
    Class used for testing RandomDivergentConnect.
    '''

    def __init__(self, N_s, N_t, C, e_min=10, threads=1):
        '''
        Sets up the experiment, and calculates expected distributions
        for later comparison with the observed distribution.

        Parameters
        ----------
            N_s    : Number of source neurons.
            N_t    : Number of target neurons.
            C      : Out-degree (number of connections per
                     source neuron).
            e_min  : Minimum expected number of observations in
                     each bin. Default is 10.
            threads: Set number of local threads. Default is 1.
        '''

        self.N_s = N_s
        self.N_t = N_t
        self.C = C
        self.e_min = e_min
        self.threads = threads

        nest.ResetKernel()
        nest.SetKernelStatus({'local_num_threads': self.threads})
        self.n_vp = nest.GetKernelStatus('total_num_virtual_procs')

        expected_degree = self.N_s * self.C / float(self.N_t)
        if expected_degree < self.e_min:
            raise RuntimeWarning(
                'Expected in-degree (%.2f) is less than e_min (%.2f).' \
                'Increase N_s*C / N_t or decrease e_min.' %
                (expected_degree, self.e_min))

        self.expected = [expected_degree] * self.N_t

    def _counter(self, x):
        '''
        Count similar elements in list.

        Parameters
        ----------
            x: Any list.

        Return values
        -------------
            list containing counts of similar elements.
        '''

        start = min(x)
        counts = [0] * self.N_s
        for elem in x:
            counts[elem - start] += 1

        return counts

    def _reset(self, msd):
        '''
        Reset the NEST kernel and set seed values.

        Parameters
        ----------
            msd: Master RNG seed.
        '''

        nest.ResetKernel()
        nest.SetKernelStatus({'local_num_threads': self.threads})
        self.n_vp = nest.GetKernelStatus('total_num_virtual_procs')

        # Set PRNG seed values:
        if msd == None:
            msd = rnd.randint(1000000)
```

```
 90         msdrange = range(msd, msd + self.n_vp)
 91         nest.SetKernelStatus({'grng_seed': msd + self.n_vp,
 92                               'rng_seeds': msdrange})
 93 
 94     def _build(self):
 95         '''Create all nodes.'''
 96 
 97         self.source_nodes = nest.Create('iaf_neuron', self.N_s)
 98         self.target_nodes = nest.Create('iaf_neuron', self.N_t)
 99 
100     def _connect(self):
101         '''Connect all nodes.'''
102 
103         nest.RandomDivergentConnect(self.source_nodes, self.target_nodes,
104                                     self.C, options={'allow_multapses': True})
105 
106     def _get_degrees(self, msd):
107         '''
108         Reset NEST, create nodes, connect them, and retrieve the
109         resulting connections.
110 
111         Parameters
112         ----------
113             msd: Master RNG seed.
114 
115         Return values
116         -------------
117             list containing the in-degrees of the target nodes.
118         '''
119 
120         self._reset(msd)
121         self._build()
122         self._connect()
123 
124         connections = nest.GetConnections(target=self.target_nodes)
125         target_connections = [conn[1] for conn in connections]
126         degrees = self._counter(target_connections)
127 
128         return degrees
129 
130     def _get_degrees_control(self, msd):
131         '''
132         Instead of using NEST, this method returns data with the expected
133         multinomial distribution.
134 
135         Parameters
136         ----------
137             msd: Master RNG seed.
138 
139         Return values
140         -------------
141             list containing the in-degrees of the target nodes.
142         '''
143 
144         if msd != None:
145             rnd.seed(msd)
146 
147         con = rnd.randint(0, self.N_t, self.N_s * self.C)
148         degrees = self._counter(con)
149 
150         return degrees
151 
152     def chi_squared_test(self, msd=None, control=False):
153         '''
154         Create a single network and compare the resulting in-degree
155         distribution with the expected distribution using Pearson's chi-squared
156         GOF test.
157 
158         Parameters
159         ----------
160             msd    : Master RNG seed.
161             control: Boolean value. If True, _get_degrees_control will
162                      be used instead of _get_degrees.
163 
164         Return values
165         -------------
166             chi-squared statistic.
167             p-value from chi-squared test.
168         '''
169 
170         if control:
171             degrees = self._get_degrees_control(msd)
172         else:
173             degrees = self._get_degrees(msd)
174 
175         # ddof: adjustment to the degrees of freedom. df = k-1-ddof
176         return scipy.stats.chisquare(numpy.array(degrees),
177                                      numpy.array(self.expected), ddof=0)
178 
179     def two_level_test(self, n_runs, start_seed=None, control=False,
180                        show_histogram=False, histogram_bins=100,
181                        show_CDF=False):
182         '''
183         Create a network and run chi-squared GOF test n_runs times.
184         Test whether resulting p-values are uniformly distributed
185         on [0, 1] using the Kolmogorov-Smirnov GOF test.
186 
187         Parameters
188         ----------
189             n_runs        : number of times to repeat chi-squared test.
190             start_seed    : First PRNG seed value.
191             control       : boolean value. If True, _get_degrees_control will be
192                             used instead of _get_degrees.
193             show_histogram: Specify whether histogram should be displayed.
```

```
194             histogram_bins: Number of histogram bins.
195             show_CDF      : Specify whether EDF should be displayed.
196 
197         Return values
198         -------------
199             KS statistic.
200             p-value from KS test.
201         '''
202 
203         self.pvalues = []
204 
205         if start_seed == None:
206             for i in range(n_runs):
207                 print 'Running test %d of %d.' % (i + 1, n_runs)
208                 chi, p = self.chi_squared_test(None, control)
209                 self.pvalues.append(p)
210         else:
211             seed_jump = self.n_vp + 1
212             end_seed = start_seed + n_runs * seed_jump
213             for seed in range(start_seed, end_seed, seed_jump):
214                 print 'Running test %d of %d.' % \
215                     (1 + (seed - start_seed) / seed_jump, n_runs)
216                 chi, p = self.chi_squared_test(seed, control)
217                 self.pvalues.append(p)
218 
219         ks, p = scipy.stats.kstest(self.pvalues, 'uniform',
220                                    alternative='two_sided')
221 
222         if show_CDF:
223             plt.figure()
224             self.pvalues.sort()
225             y = [(i + 1.) / len(self.pvalues)
226                  for i in range(len(self.pvalues))]
227             plt.step([0.0] + self.pvalues + [1.0], [0.0] + y + [1.0])
228             plt.xlabel('P-values')
229             plt.ylabel('Empirical distribution function')
230 
231         if show_histogram:
232             plt.figure()
233             plt.hist(self.pvalues, bins=histogram_bins)
234             plt.xlabel('P-values')
235             plt.ylabel('Frequency')
236 
237         if show_CDF or show_histogram:
238             plt.show(block=True)
239 
240         return ks, p
241 
242 
243 if __name__ == '__main__':
244     test = RDC_tester(N_s=1000, N_t=1000, C=1000)
245     ks, p = test.two_level_test(n_runs=1000, start_seed=0)
246     print 'KS test statistic:', ks
247     print 'p-value of KS-test of uniformity:', p
```

Appendix D

Automated test suite for random convergent and divergent connections

```
'''
@author: Daniel Hjertholm

Unittests for RandomConvergentConnect and RandomDivergentConnect.
'''

import unittest

from test_RCC import RCC_tester
from test_RDC import RDC_tester


class RCDCTestCase(unittest.TestCase):
    '''Statistical tests for Random{Con,Di}vergentConnect.'''

    def setUp(self):
        '''Set test parameters and critical values.'''

        # Small network
        self.N_small = 10   # Number of nodes
        self.C_small = 100  # Average number of connections per node
        self.n_small = 1000 # Number of times to repeat test

        # Medium sized network
        self.N_medium = 100 # Number of nodes
        self.C_medium = 100 # Average number of connections per node
        self.n_medium = 100 # Number of times to repeat test

        # Large network
        self.N_large = 1000 # Number of nodes
        self.C_large = 1000 # Average number of connections per node
        self.n_large = 100  # Number of times to repeat test

        # Critical values
        self.alpha1_lower = 0.025
        self.alpha1_upper = 0.975
        self.alpha2 = 0.05

    def adaptive_test(self, test, n_runs):
        '''
        Create a single network using Random{Con/Di}vergentConnect
        and run a chi-squared GOF test on the connection distribution.
        If the result is extreme (high or low), run a two-level test.

        Parameters
        ----------
            test  : Instance of RCC_tester or RDC_tester class.
            n_runs: If chi-square test fails, test is repeated n_runs times,
                    and the KS test is used to analyze results.

        Return values
        -------------
            boolean value. True if test was passed, False otherwise.
        '''

        chi, p = test.chi_squared_test(msd=None)

        if self.alpha1_lower < p < self.alpha1_upper:
            return True
        else:
            ks, p = test.two_level_test(n_runs=n_runs, start_seed=None)
            return True if p > self.alpha2 else False

    def test_RCC_small(self):
        '''Statistical test of RandomConvergentConnect with a small network'''

        test = RCC_tester(N_s=self.N_small, N_t=self.N_small, C=self.C_small)
        passed = self.adaptive_test(test, n_runs=self.n_small)
        self.assertTrue(passed, 'RandomConvergentConnect did not ' \
                        'pass the statistical test procedure.')

    def test_RCC_large(self):
        '''Statistical test of RandomConvergentConnect with a large network'''

        test = RCC_tester(N_s=self.N_large, N_t=self.N_large, C=self.C_large)
        passed = self.adaptive_test(test, n_runs=self.n_large)
        self.assertTrue(passed, 'RandomConvergentConnect did not ' \
                        'pass the statistical test procedure.')

    def test_RCC_threaded(self):
        '''Statistical test of RandomConvergentConnect with 4 threads'''

```

```
83        test = RCC_tester(N_s=self.N_medium, N_t=self.N_medium, C=self.C_medium,
84                          threads=4)
85        passed = self.adaptive_test(test, n_runs=self.n_medium)
86        self.assertTrue(passed, 'RandomConvergentConnect did not ' \
87                        'pass the statistical test procedure.')
88
89    def test_RDC_small(self):
90        '''Statistical test of RandomDivergentConnect with a small network'''
91
92        test = RDC_tester(N_s=self.N_small, N_t=self.N_small, C=self.C_small)
93        passed = self.adaptive_test(test, n_runs=self.n_small)
94        self.assertTrue(passed, 'RandomDivergentConnect did not ' \
95                        'pass the statistical test procedure.')
96
97    def test_RDC_large(self):
98        '''Statistical test of RandomDivergentConnect with a large network'''
99
100       test = RDC_tester(N_s=self.N_large, N_t=self.N_large, C=self.C_large)
101       passed = self.adaptive_test(test, n_runs=self.n_large)
102       self.assertTrue(passed, 'RandomDivergentConnect did not ' \
103                       'pass the statistical test procedure.')
104
105   def test_RDC_threaded(self):
106       '''Statistical test of RandomDivergentConnect with 4 threads'''
107
108       test = RDC_tester(N_s=self.N_medium, N_t=self.N_medium, C=self.C_medium,
109                         threads=4)
110       passed = self.adaptive_test(test, n_runs=self.n_medium)
111       self.assertTrue(passed, 'RandomDivergentConnect did not ' \
112                       'pass the statistical test procedure.')
113
114
115 def suite():
116     suite = unittest.makeSuite(RCDCTestCase, 'test')
117     return suite
118
119
120 if __name__ == '__main__':
121     runner = unittest.TextTestRunner(verbosity=2)
122     runner.run(suite())
```

Appendix E

Test script for 2D spatially structured network

```
'''
@author: Daniel Hjertholm

Tests for ConnectLayers.
'''

import numpy as np
import numpy.random as rnd
import scipy.integrate
import scipy.stats
import matplotlib.pyplot as plt
import nest
import nest.topology as topo


class ConnectLayers2D_tester(object):
    '''Class used for testing ConnectLayers.'''

    def __init__(self, L, N, kernel_name, kernel_params=None,
                 source_pos=None, msd=None, threads=1):
        '''
        Initialize an ConnectLayers2D_tester object.

        Sets up the experiment, and defines network layer and
        connection parameters .

        Parameters
        ----------
            L            : Side length of square layer.
            N            : Number of  nodes.
            kernel_name  : Name of distance dependent probability
                           function (kernel) to test.
            kernel_params: Parameters for kernel function. If
                           omitted, sensible defaults are calculated
                           based on layer size.
            source_pos   : Source node position. Default is center.
            msd          : Master PRNG seed. Default is None.
            threads      : Number of local threads. Default is 1.
        '''

        self.threads = threads
        self.msd = msd
        self.L = float(L)
        self.N = N
        self.kernel_name = kernel_name

        kernels = {
            'constant': self._constant,
            'linear': self._linear,
            'exponential': self._exponential,
            'gaussian': self._gauss}
        default_params = {
            'constant': 1.0,
            'linear': {'a':-np.sqrt(2) / self.L, 'c': 1.0},
            'exponential': {'a':1.0, 'c': 0.0,
                            'tau':-self.L /
                                  (np.sqrt(2) * np.log((.1 - 0) / 1))},
            'gaussian': {'p_center': 1., 'sigma': self.L / 4.,
                         'mean': 0., 'c': 0.}}

        self.params = default_params[kernel_name]
        if kernel_params is not None:
            if self.kernel_name == 'constant':
                self.params = kernel_params
            else:
                self.params.update(kernel_params)

        self.kernel_func = kernels[kernel_name]

        if source_pos is None: source_pos = (0., 0.)
        self.ldict_s = {'elements': 'iaf_neuron', 'positions': [source_pos],
                        'extent': [self.L, self.L], 'edge_wrap': True}

        if msd is not None:
            rnd.seed(msd)
        x = rnd.uniform(-self.L / 2., self.L / 2., self.N)
        y = rnd.uniform(-self.L / 2., self.L / 2., self.N)
        pos = zip(x, y)
        self.ldict_t = {'elements': 'iaf_neuron', 'positions': pos,
                        'extent': [self.L, self.L], 'edge_wrap': True}
        self.mask = {'rectangular': {'lower_left': [-self.L / 2.,
                                                     - self.L / 2.],
                                     'upper_right': [self.L / 2.,
                                                      self.L / 2.]}}
        if kernel_name == 'constant':
            self.kernel = self.params
        else:
            self.kernel = {self.kernel_name: self.params}
        self.conndict = {'connection_type': 'divergent',
```

```
90                            'mask': self.mask, 'kernel': self.kernel}
91
92      def _constant(self, D):
93          '''Constant kernel function.'''
94
95          return self.params
96
97      def _linear(self, D):
98          '''Linear kernel function.'''
99
100         return self.params['c'] + self.params['a'] * D
101
102     def _exponential(self, D):
103         '''Exponential kernel function.'''
104
105         return (self.params['c'] + self.params['a'] *
106                 np.e ** (-D / self.params['tau']))
107
108     def _gauss(self, D):
109         '''Gaussian kernel function.'''
110
111         return (self.params['c'] + self.params['p_center'] *
112                 np.e ** -((D - self.params['mean']) ** 2 /
113                           (2. * self.params['sigma'] ** 2)))
114
115     def _pdf(self, D):
116         '''
117         Unnormalized probability density function (PDF).
118
119         Parameters
120         ----------
121             D: Distance in interval [0, L/sqrt(2)].
122
123         Return values
124         -------------
125             Unnormalized PDF at distance D.
126         '''
127
128         if D <= self.L / 2.:
129             return (max(0., min(1., self.kernel_func(D))) * np.pi * D)
130         elif self.L / 2. < D <= self.L / np.sqrt(2):
131             return (max(0., min(1., self.kernel_func(D))) *
132                     D * (np.pi - 4. * np.arccos(self.L / (D * 2.))))
133         else:
134             return 0.
135
136     def _cdf(self, D):
137         '''
138         Normalized cumulative distribution function (CDF).
139
140         Parameters
141         ----------
142             D: Iterable of distances in interval [0, L/sqrt(2)].
143
144         Return values
145         -------------
146             List of CDF(d) for each distance d in D.
147         '''
148
149         cdf = []
150         last_d = 0.
151         for d in D:
152             cdf.append(scipy.integrate.quad(self._pdf, last_d, d)[0])
153             last_d = d
154
155         cdf = np.cumsum(cdf)
156
157         top = scipy.integrate.quad(self._pdf, 0, self.L / np.sqrt(2))[0]
158         normed_cdf = cdf / top
159
160         # Stored in case the CDF is to be plotted later.
161         self.cdf_list = normed_cdf
162
163         return normed_cdf
164
165     def _reset(self):
166         '''Reset the NEST kernel and set PRNG seed values.'''
167
168         nest.ResetKernel()
169         nest.SetKernelStatus({'local_num_threads': self.threads})
170         self.n_vp = nest.GetKernelStatus('total_num_virtual_procs')
171
172         # Set PRNG seed values:
173         if self.msd is None:
174             msd = rnd.randint(1000000)
175         else:
176             msd = self.msd + 1 # The first was used in __init__.
177         msdrange = range(msd, msd + self.n_vp)
178         nest.SetKernelStatus({'grng_seed': msd + self.n_vp,
179                               'rng_seeds': msdrange})
180
181     def _build(self):
182         '''Create layers.'''
183
184         self.ls = topo.CreateLayer(self.ldict_s)
185         self.lt = topo.CreateLayer(self.ldict_t)
186         self.driver = topo.FindCenterElement(self.ls)
187
188     def _connect(self):
189         '''Connect layers.'''
190
191         topo.ConnectLayers(self.ls, self.lt, self.conndict)
192
193     def _get_distances(self):
```

```
194         '''
195         Create and connect layers, and get distances to connected nodes.
196 
197         Return values
198         -------------
199             Ordered list of distances to connected nodes.
200         '''
201 
202         self._reset()
203         self._build()
204         self._connect()
205 
206         connections = nest.GetConnections(source=self.driver)
207         target_nodes = [conn[1] for conn in connections]
208         if len(target_nodes) == 0: return None, None
209         dist = topo.Distance(self.driver, target_nodes)
210         dist.sort()
211 
212         return dist
213 
214     def _get_distances_control(self):
215         '''
216         Instead of using NEST's actual connection algorithm, this
217         method creates data with the expected distribution.
218 
219         Return values
220         -------------
221             Ordered list of distances to connected nodes.
222         '''
223 
224         self._reset()
225         self._build()
226 
227         dist = topo.Distance(self.driver, nest.GetLeaves(self.lt)[0])
228         dist = [d for d in dist if rnd.uniform() < self.kernel_func(d)]
229         dist.sort()
230 
231         return dist
232 
233     def ks_test(self, control=False, alternative='two_sided',
234                 show_network=False, show_CDF=False, show_PDF=False,
235                 histogram_bins=100):
236         '''
237         Perform a Kolmogorov-Smirnov GOF test on the distribution
238         of distances to connected nodes.
239 
240         Parameters
241         ----------
242             control       : Boolean value. If True, _get_distances_control
243                             will be used instead of _get_distances.
244             alternative   : Alternative hypothesis. 'two_sided' (default),
245                             'less' or 'greater'.
246             show_network  : Specify whether network plot should be displayed.
247             show_CDF      : Specify whether CDF should be displayed.
248             show_PDF      : Specify whether PDF should be displayed.
249             histogram_bins: Number of histogram bins for PDF plot.
250 
251         Return values
252         -------------
253             KS statistic.
254             p-value from KS test.
255         '''
256 
257         if control:
258             dist = self._get_distances_control()
259         else:
260             dist = self._get_distances()
261 
262         ks, p = scipy.stats.kstest(dist, self._cdf,
263                                    alternative=alternative)
264 
265         if show_network and not control:
266             # Adjust size of nodes in plot based on number of nodes.
267             nodesize = max(1, int(round(111. / 11 - self.N / 11000.)))
268 
269             fig = topo.PlotLayer(self.lt, nodesize=nodesize,
270                                  nodecolor='grey')
271 
272             # Only the gaussian kernel can be plotted.
273             if self.kernel_name == 'gaussian':
274                 topo.PlotTargets(self.driver, self.lt, fig=fig,
275                                  mask=self.mask, kernel=self.kernel,
276                                  mask_color='purple',
277                                  kernel_color='blue',
278                                  src_size=50, src_color='black',
279                                  tgt_size=nodesize, tgt_color='red')
280             else:
281                 topo.PlotTargets(self.driver, self.lt, fig=fig,
282                                  mask=self.mask, mask_color='purple',
283                                  src_size=50, src_color='black',
284                                  tgt_size=nodesize, tgt_color='red')
285 
286         if show_CDF:
287             plt.figure()
288             plt.plot(dist, self.cdf_list, '-', color='black', linewidth=3,
289                      label='Theory', zorder=1)
290             y = [(i + 1.) / len(dist) for i in range(len(dist))]
291             plt.step([0.0] + dist, [0.0] + y, color='red',
292                      linewidth=1, label='Empirical', zorder=2)
293             plt.ylim(0, 1)
294 
295             plt.xlabel('Distance')
296             plt.ylabel('CDF')
297             plt.legend(loc='center right')
```

```
298
299         if show_PDF:
300             plt.figure()
301             x = np.linspace(0, self.L / np.sqrt(2), 1000)
302             area = scipy.integrate.quad(self._pdf, 0, self.L / np.sqrt(2))[0]
303             y = np.array([self._pdf(D) for D in x]) / area
304             plt.plot(x, y, color='black', linewidth=3, label='Theory', zorder=1)
305             plt.hist(dist, bins=histogram_bins, histtype='step',
306                      linewidth=1, normed=True, color='red',
307                      label='Empirical', zorder=2)
308             plt.ylim(ymin=0.)
309
310             plt.xlabel('Distance')
311             plt.ylabel('PDF')
312             plt.legend(loc='center right')
313
314         if show_CDF or show_PDF or show_network:
315             plt.show(block=True)
316
317         return ks, p
318
319     def z_test(self, control=False):
320         '''
321         Perform a Z-test on the total number of connections.
322
323         Parameters
324         ---------
325             control: Boolean value. If True, _get_distances_control will be used
326             instead of _get_distances.
327
328         Return values
329         -------------
330             Standard score (z-score).
331             Two-sided p-value.
332         '''
333
334         if control:
335             num = len(self._get_distances_control())
336         else:
337             num = len(self._get_distances())
338
339         dist = topo.Distance(self.driver, nest.GetLeaves(self.lt)[0])
340         ps = ([max(0., min(1., self.kernel_func(D))) for D in dist])
341         exp = sum(ps)
342         var = sum([p * (1. - p) for p in ps])
343         if var == 0: return np.nan, 1.0
344         sd = np.sqrt(var)
345         z = abs((num - exp) / sd)
346         p = 2. * (1. - scipy.stats.norm.cdf(z))
347
348         return z, p
349
350
351 if __name__ == '__main__':
352     test = ConnectLayers2D_tester(L=1.0, N=100000,
353                                   kernel_name='gaussian', msd=0)
354     ks, p = test.ks_test(show_network=True, show_PDF=True, show_CDF=True)
355     print 'KS test statistic:', ks
356     print 'p-value of KS-test:', p
357     z, p = test.z_test()
358     print 'Z-score:', z
359     print 'p-value of Z-test:', p
```

Appendix F

Test script for 3D spatially structured network

```
'''
@author: Daniel Hjertholm

Tests for ConnectLayers.
'''

import numpy as np
import numpy.random as rnd
import scipy.integrate
import scipy.stats
import matplotlib.pyplot as plt
import nest
import nest.topology as topo


class ConnectLayers3D_tester(object):
    '''Class used for testing ConnectLayers.'''

    def __init__(self, L, N, kernel_name, kernel_params=None,
                 source_pos=None, msd=None, threads=1):
        '''
        Initialize an ConnectLayers3D_tester object.

        Sets up the experiment, and defines network layer and
        connection parameters .

        Parameters
        ----------
            L              : Side length of cubic layer.
            N              : Number of  nodes.
            kernel_name    : Name of distance dependent probability
                             function (kernel) to test.
            kernel_params  : Parameters for kernel function. If
                             omitted, sensible defaults are calculated
                             based on layer size.
            source_pos     : Source node position. Default is center.
            msd            : Master PRNG seed. Default is None.
            threads        : Number of local threads. Default is 1.
        '''

        self.threads = threads
        self.msd = msd
        self.L = float(L)
        self.N = N
        self.kernel_name = kernel_name

        kernels = {
            'constant': self._constant,
            'linear': self._linear,
            'exponential': self._exponential,
            'gaussian': self._gauss}
        default_params = {
            'constant': 1.0,
            'linear': {'a':-2.0 / (np.sqrt(3) * self.L), 'c': 1.0},
            'exponential': {'a':1.0, 'c': 0.0,
                            'tau':-self.L * np.sqrt(3) /
                                  (2.0 * np.log((.1 - 0) / 1))},
            'gaussian': {'p_center': 1., 'sigma': self.L / 4.,
                         'mean': 0., 'c': 0.}}

        self.params = default_params[kernel_name]
        if kernel_params is not None:
            if self.kernel_name == 'constant':
                self.params = kernel_params
            else:
                self.params.update(kernel_params)

        self.kernel_func = kernels[kernel_name]

        if source_pos is None: source_pos = (0., 0., 0.)
        self.ldict_s = {'elements': 'iaf_neuron', 'positions': [source_pos],
                        'extent': [self.L, self.L, self.L], 'edge_wrap': True}

        if msd is not None:
            rnd.seed(msd)
        x = rnd.uniform(-self.L / 2., self.L / 2., self.N)
        y = rnd.uniform(-self.L / 2., self.L / 2., self.N)
        z = rnd.uniform(-self.L / 2., self.L / 2., self.N)
        pos = zip(x, y, z)
        self.ldict_t = {'elements': 'iaf_neuron', 'positions': pos,
                        'extent': [self.L, self.L, self.L], 'edge_wrap': True}
        self.mask = {'box': {'lower_left': [-self.L / 2.,
                                            - self.L / 2.,
                                            - self.L / 2.],
                             'upper_right': [self.L / 2.,
                                             self.L / 2.,
                                             self.L / 2.]}}
        if kernel_name == 'constant':
            self.kernel = self.params
```

```
 90         else:
 91             self.kernel = {self.kernel_name: self.params}
 92         self.conndict = {'connection_type': 'divergent',
 93                          'mask': self.mask, 'kernel': self.kernel}
 94 
 95     def _constant(self, D):
 96         '''Constant kernel function.'''
 97 
 98         return self.params
 99 
100     def _linear(self, D):
101         '''Linear kernel function.'''
102 
103         return self.params['c'] + self.params['a'] * D
104 
105     def _exponential(self, D):
106         '''Exponential kernel function.'''
107 
108         return (self.params['c'] + self.params['a'] *
109                 np.e ** (-D / self.params['tau']))
110 
111     def _gauss(self, D):
112         '''Gaussian kernel function.'''
113 
114         return (self.params['c'] + self.params['p_center'] *
115                 np.e ** -((D - self.params['mean']) ** 2 /
116                           (2. * self.params['sigma'] ** 2)))
117 
118     def _pdf(self, D):
119         '''
120         Unnormalized probability density function (PDF).
121 
122         Parameters
123         ----------
124             D: Distance in interval [0, L/sqrt(2)].
125 
126         Return values
127         -------------
128             Unnormalized PDF at distance D.
129         '''
130 
131         if D <= self.L / 2.:
132             return (max(0., min(1., self.kernel_func(D))) *
133                     4. * np.pi * D ** 2.)
134         elif self.L / 2. < D <= self.L / np.sqrt(2):
135             return (max(0., min(1., self.kernel_func(D))) *
136                     2. * np.pi * D * (3. * self.L - 4. * D))
137         elif self.L / np.sqrt(2) < D <= self.L * np.sqrt(3) / 2.:
138             A = 4. * np.pi * D ** 2.
139             C = 2. * np.pi * D * (D - self.L / 2.)
140             alpha = np.arcsin(1. / np.sqrt(2. - self.L ** 2. / (2. * D ** 2.)))
141             beta = np.pi / 2.
142             gamma = np.arcsin(np.sqrt((1. - .5 * (self.L / D) ** 2.) /
143                                       (1. - .25 * (self.L / D) ** 2.)))
144             T = D ** 2. * (alpha + beta + gamma - np.pi)
145             return (max(0., min(1., self.kernel_func(D))) *
146                     (A + 6. * C * (-1. + 4. * gamma / np.pi) - 48. * T))
147         else:
148             return 0.
149 
150     def _cdf(self, D):
151         '''
152         Normalized cumulative distribution function (CDF).
153 
154         Parameters
155         ----------
156             D: Iterable of distances in interval [0, L/sqrt(2)].
157 
158         Return values
159         -------------
160             List of CDF(d) for each distance d in D.
161         '''
162 
163         cdf = []
164         last_d = 0.
165         for d in D:
166             cdf.append(scipy.integrate.quad(self._pdf, last_d, d)[0])
167             last_d = d
168 
169         cdf = np.cumsum(cdf)
170 
171         top = scipy.integrate.quad(self._pdf, 0, self.L * np.sqrt(3) / 2)[0]
172         normed_cdf = cdf / top
173 
174         # Stored in case the CDF is to be plotted later.
175         self.cdf_list = normed_cdf
176 
177         return normed_cdf
178 
179     def _reset(self):
180         '''Reset the NEST kernel and set PRNG seed values.'''
181 
182         nest.ResetKernel()
183         nest.SetKernelStatus({'local_num_threads': self.threads})
184         self.n_vp = nest.GetKernelStatus('total_num_virtual_procs')
185 
186         # Set PRNG seed values:
187         if self.msd is None:
188             msd = rnd.randint(1000000)
189         else:
190             msd = self.msd + 1 # The first was used in __init__.
191         msdrange = range(msd, msd + self.n_vp)
192         nest.SetKernelStatus({'grng_seed': msd + self.n_vp,
193                               'rng_seeds': msdrange})
```

```
194
195     def _build(self):
196         '''Create layers.'''
197
198         self.ls = topo.CreateLayer(self.ldict_s)
199         self.lt = topo.CreateLayer(self.ldict_t)
200         self.driver = topo.FindCenterElement(self.ls)
201
202     def _connect(self):
203         '''Connect layers.'''
204
205         topo.ConnectLayers(self.ls, self.lt, self.conndict)
206
207     def _get_distances(self):
208         '''
209         Create and connect layers, and get distances to connected nodes.
210
211         Return values
212         -------------
213             Ordered list of distances to connected nodes.
214         '''
215
216         self._reset()
217         self._build()
218         self._connect()
219
220         connections = nest.GetConnections(source=self.driver)
221         target_nodes = [conn[1] for conn in connections]
222         if len(target_nodes) == 0: return None, None
223         dist = topo.Distance(self.driver, target_nodes)
224         dist.sort()
225
226         return dist
227
228     def _get_distances_control(self):
229         '''
230         Instead of using NEST's actual connection algorithm, this
231         method creates data with the expected distribution.
232
233         Return values
234         -------------
235             Ordered list of distances to connected nodes.
236         '''
237
238         self._reset()
239         self._build()
240
241         dist = topo.Distance(self.driver, nest.GetLeaves(self.lt)[0])
242         dist = [d for d in dist if rnd.uniform() < self.kernel_func(d)]
243         dist.sort()
244
245         return dist
246
247     def ks_test(self, control=False, alternative='two_sided',
248                 show_network=False, show_CDF=False, show_PDF=False,
249                 histogram_bins=100):
250         '''
251         Perform a Kolmogorov-Smirnov GOF test on the distribution
252         of distances to connected nodes.
253
254         Parameters
255         ----------
256             control       : Boolean value. If True, _get_distances_control
257                             will be used instead of _get_distances.
258             alternative   : Alternative hypothesis. 'two_sided' (default),
259                             'less' or 'greater'.
260             show_network  : Specify whether network plot should be displayed.
261             show_CDF      : Specify whether CDF should be displayed.
262             show_PDF      : Specify whether PDF should be displayed.
263             histogram_bins: Number of histogram bins for PDF plot.
264
265         Return values
266         -------------
267             KS statistic.
268             p-value from KS test.
269         '''
270
271         if control:
272             dist = self._get_distances_control()
273         else:
274             dist = self._get_distances()
275
276         ks, p = scipy.stats.kstest(dist, self._cdf,
277                                    alternative=alternative)
278
279         if show_network and not control:
280             # Adjust size of nodes in plot based on number of nodes.
281             nodesize = max(1, int(round(111. / 11 - self.N / 11000.)))
282
283             fig = topo.PlotLayer(self.lt, nodesize=nodesize,
284                                  nodecolor='grey')
285
286             # Only the gaussian kernel can be plotted.
287             if self.kernel_name == 'gaussian':
288                 topo.PlotTargets(self.driver, self.lt, fig=fig,
289                                  mask=self.mask, kernel=self.kernel,
290                                  mask_color='purple',
291                                  kernel_color='blue',
292                                  src_size=50, src_color='black',
293                                  tgt_size=nodesize, tgt_color='red')
294             else:
295                 topo.PlotTargets(self.driver, self.lt, fig=fig,
296                                  mask=self.mask, mask_color='purple',
297                                  src_size=50, src_color='black',
```

```
298                                 tgt_size=nodesize, tgt_color='red')
299 
300         if show_CDF:
301             plt.figure()
302             plt.plot(dist, self.cdf_list, '-', color='black', linewidth=3,
303                      label='Theory', zorder=1)
304             y = [(i + 1.) / len(dist) for i in range(len(dist))]
305             plt.step([0.0] + dist, [0.0] + y, color='red',
306                      linewidth=1, label='Empirical', zorder=2)
307             plt.ylim(0, 1)
308 
309             plt.xlabel('Distance')
310             plt.ylabel('CDF')
311             plt.legend(loc='center right')
312 
313         if show_PDF:
314             plt.figure()
315             x = np.linspace(0, self.L * np.sqrt(3) / 2, 1000)
316             area = scipy.integrate.quad(self._pdf, 0,
317                                         self.L * np.sqrt(3) / 2)[0]
318             y = np.array([self._pdf(D) for D in x]) / area
319             plt.plot(x, y, color='black', linewidth=3, label='Theory', zorder=1)
320             plt.hist(dist, bins=histogram_bins, histtype='step',
321                      linewidth=1, normed=True, color='red',
322                      label='Empirical', zorder=2)
323             plt.ylim(ymin=0.)
324 
325             plt.xlabel('Distance')
326             plt.ylabel('PDF')
327             plt.legend(loc='center right')
328 
329         if show_CDF or show_PDF or show_network:
330             plt.show(block=True)
331 
332         return ks, p
333 
334     def z_test(self, control=False):
335         '''
336         Perform a Z-test on the total number of connections.
337 
338         Parameters
339         ---------
340             control: Boolean value. If True, _get_distances_control will be used
341             instead of _get_distances.
342 
343         Return values
344         -------------
345             Standard score (z-score).
346             Two-sided p-value.
347         '''
348 
349         if control:
350             num = len(self._get_distances_control())
351         else:
352             num = len(self._get_distances())
353 
354         dist = topo.Distance(self.driver, nest.GetLeaves(self.lt)[0])
355         ps = ([max(0., min(1., self.kernel_func(D))) for D in dist])
356         exp = sum(ps)
357         var = sum([p * (1. - p) for p in ps])
358         if var == 0: return np.nan, 1.0
359         sd = np.sqrt(var)
360         z = abs((num - exp) / sd)
361         p = 2. * (1. - scipy.stats.norm.cdf(z))
362 
363         return z, p
364 
365 
366 if __name__ == '__main__':
367     test = ConnectLayers3D_tester(L=1.0, N=100000,
368                                   kernel_name='gaussian', msd=0)
369     ks, p = test.ks_test(show_network=True, show_PDF=True, show_CDF=True)
370     print 'KS test statistic:', ks
371     print 'p-value of KS-test:', p
372     z, p = test.z_test()
373     print 'Z-score:', z
374     print 'p-value of Z-test:', p
```

Appendix G

Automated test suite for spatially structured networks

```
'''
@author: Daniel Hjertholm

Unittests for 2D and 3D spatially structured networks.
'''

import unittest
import scipy.stats

from test_2D import ConnectLayers2D_tester
from test_3D import ConnectLayers3D_tester


class ConnectLayersTestCase(unittest.TestCase):
    '''Statistical tests for ConnectLayers.'''

    def setUp(self):
        '''Set test parameters and critical values.'''

        self.N = 100000   # Number of nodes
        self.L = 1.0      # Layer size.
        self.n_runs = 100 # Number of times to repeat test

        # Critical values
        self.alpha1 = 0.01
        self.alpha2 = 0.01

    def ks_test(self, tester_class, kernel_name, kernel_params={},
                source_pos=None, threads=1):
        '''
        Create a single network using ConnectLayers, and perform a
        Kolmogorov-Smirnov  (KS) test on the distribution of source-target
        distances. If the result is suspicious, the test is repeated n_runs
        times, and the resulting p-values are compared with the expected uniform
        distribution using the KS test.

        Parameters
        ----------
            tester_class : ConnectLayers2D_tester or ConnectLayers3D_tester.
            kernel_name  : Name of distance dependent probability
                           function (kernel) to test.
            kernel_params: Parameters for kernel function. Optional.
            source_pos   : Source node position. Optional. Default is center.
            threads      : Number of local threads. Optional. Default is 1.

        Return values
        -------------
            boolean value. True if test was passed, False otherwise.
        '''

        test = tester_class(L=self.L, N=self.N, kernel_name=kernel_name,
                            kernel_params=kernel_params,
                            source_pos=source_pos, msd=None)
        ks, p = test.ks_test()

        if p > self.alpha1:
            return True
        else:
            ps = []
            print ''
            for i in range(self.n_runs):
                print 'Running test %d of %d.' % (i + 1, self.n_runs)
                test = tester_class(L=self.L, N=self.N,
                                    kernel_name=kernel_name,
                                    kernel_params=kernel_params,
                                    source_pos=source_pos, msd=None)
                ps.append(test.ks_test()[1])
            ks, p = scipy.stats.kstest(ps, 'uniform', alternative='two_sided')
            return True if p > self.alpha2 else False

    def z_test(self, tester_class, kernel_name, kernel_params={},
               source_pos=None, threads=1):
        '''
        Create a single network using ConnectLayers, and perform a Z-test on the
        total connection count. If the result is suspicious, the test is
        repeated n_runs times, and the resulting p-values are compared with the
        expected uniform distribution using the KS test.

        Parameters
        ----------
            tester_class : ConnectLayers2D_tester or ConnectLayers3D_tester.
            kernel_name  : Name of distance dependent probability
                           function (kernel) to test.
            kernel_params: Parameters for kernel function. Optional.
            source_pos   : Source node position. Optional. Default is center.
            threads      : Number of local threads. Optional. Default is 1.

        Return values
        -------------
```

```
90              boolean value. True if test was passed, False otherwise.
91          '''
92  
93          test = tester_class(L=self.L, N=self.N, kernel_name=kernel_name,
94                              kernel_params=kernel_params,
95                              source_pos=source_pos, msd=None)
96          z, p = test.z_test()
97  
98          if p > self.alpha1:
99              return True
100         else:
101             ps = []
102             print ''
103             for i in range(self.n_runs):
104                 print 'Running test %d of %d.' % (i + 1, self.n_runs)
105                 test = tester_class(L=self.L, N=self.N,
106                                     kernel_name=kernel_name,
107                                     kernel_params=kernel_params,
108                                     source_pos=source_pos, msd=None)
109                 ps.append(test.z_test()[1])
110             z, p = scipy.stats.kstest(ps, 'uniform', alternative='two_sided')
111             return True if p > self.alpha2 else False
112 
113     def test_2D_constant_ks(self):
114         '''KS test performed on source-target node distances'''
115 
116         self.assertTrue(self.ks_test(ConnectLayers2D_tester, 'constant',
117                                      kernel_params=0.5),
118                         'ConnectLayers failed to pass the KS test.')
119 
120     def test_2D_constant_z(self):
121         '''Z-test performed on total connection count'''
122 
123         self.assertTrue(self.z_test(ConnectLayers2D_tester, 'constant',
124                                     kernel_params=0.5),
125                         'ConnectLayers failed to pass the Z-test')
126 
127     def test_2D_linear_ks(self):
128         '''KS test performed on source-target node distances'''
129 
130         self.assertTrue(self.ks_test(ConnectLayers2D_tester, 'linear'),
131                         'ConnectLayers failed to pass the KS test.')
132 
133     def test_2D_linear_z(self):
134         '''Z-test performed on total connection count'''
135 
136         self.assertTrue(self.z_test(ConnectLayers2D_tester, 'linear'),
137                         'ConnectLayers failed to pass the Z-test')
138 
139     def test_2D_exponential_ks(self):
140         '''KS test performed on source-target node distances'''
141 
142         self.assertTrue(self.ks_test(ConnectLayers2D_tester, 'exponential'),
143                         'ConnectLayers failed to pass the KS test.')
144 
145     def test_2D_exponential_z(self):
146         '''Z-test performed on total connection count'''
147 
148         self.assertTrue(self.z_test(ConnectLayers2D_tester, 'exponential'),
149                         'ConnectLayers failed to pass the Z-test')
150 
151     def test_2D_gaussian_ks(self):
152         '''KS test performed on source-target node distances'''
153 
154         self.assertTrue(self.ks_test(ConnectLayers2D_tester, 'gaussian'),
155                         'ConnectLayers failed to pass the KS test.')
156 
157     def test_2D_gaussian_z(self):
158         '''Z-test performed on total connection count'''
159 
160         self.assertTrue(self.z_test(ConnectLayers2D_tester, 'gaussian'),
161                         'ConnectLayers failed to pass the Z-test')
162 
163     def test_2D_linear_shifted_ks(self):
164         '''KS test performed on source-target node distances'''
165 
166         self.assertTrue(self.ks_test(ConnectLayers2D_tester, 'linear',
167                                      source_pos=(self.L / 4., self.L / 4.)),
168                         'ConnectLayers failed to pass the KS test.')
169 
170     def test_2D_linear_shifted_z(self):
171         '''Z-test performed on total connection count'''
172 
173         self.assertTrue(self.z_test(ConnectLayers2D_tester, 'linear',
174                                     source_pos=(self.L / 4., self.L / 4.)),
175                         'ConnectLayers failed to pass the Z-test')
176 
177     def test_2D_linear_multithread_ks(self):
178         '''KS test performed on source-target node distances'''
179 
180         self.assertTrue(self.ks_test(ConnectLayers2D_tester, 'linear',
181                                      threads=4),
182                         'ConnectLayers failed to pass the KS test.')
183 
184     def test_2D_linear_multithread_z(self):
185         '''Z-test performed on total connection count'''
186 
187         self.assertTrue(self.z_test(ConnectLayers2D_tester, 'linear',
188                                     threads=4),
189                         'ConnectLayers failed to pass the Z-test')
190 
191     def test_3D_constant_ks(self):
192         '''KS test performed on source-target node distances'''
193 
```

```python
194         self.assertTrue(self.ks_test(ConnectLayers3D_tester, 'constant',
195                                      kernel_params=0.5),
196                         'ConnectLayers failed to pass the KS test.')
197 
198     def test_3D_constant_z(self):
199         '''Z-test performed on total connection count'''
200 
201         self.assertTrue(self.z_test(ConnectLayers3D_tester, 'constant',
202                                     kernel_params=0.5),
203                         'ConnectLayers failed to pass the Z-test')
204 
205     def test_3D_linear_ks(self):
206         '''KS test performed on source-target node distances'''
207 
208         self.assertTrue(self.ks_test(ConnectLayers3D_tester, 'linear'),
209                         'ConnectLayers failed to pass the KS test.')
210 
211     def test_3D_linear_z(self):
212         '''Z-test performed on total connection count'''
213 
214         self.assertTrue(self.z_test(ConnectLayers3D_tester, 'linear'),
215                         'ConnectLayers failed to pass the Z-test')
216 
217     def test_3D_exponential_ks(self):
218         '''KS test performed on source-target node distances'''
219 
220         self.assertTrue(self.ks_test(ConnectLayers3D_tester, 'exponential'),
221                         'ConnectLayers failed to pass the KS test.')
222 
223     def test_3D_exponential_z(self):
224         '''Z-test performed on total connection count'''
225 
226         self.assertTrue(self.z_test(ConnectLayers3D_tester, 'exponential'),
227                         'ConnectLayers failed to pass the Z-test')
228 
229     def test_3D_gaussian_ks(self):
230         '''KS test performed on source-target node distances'''
231 
232         self.assertTrue(self.ks_test(ConnectLayers3D_tester, 'gaussian'),
233                         'ConnectLayers failed to pass the KS test.')
234 
235     def test_3D_gaussian_z(self):
236         '''Z-test performed on total connection count'''
237 
238         self.assertTrue(self.z_test(ConnectLayers3D_tester, 'gaussian'),
239                         'ConnectLayers failed to pass the Z-test')
240 
241     def test_3D_linear_shifted_ks(self):
242         '''KS test performed on source-target node distances'''
243 
244         self.assertTrue(self.ks_test(ConnectLayers3D_tester, 'linear',
245             source_pos=(self.L / 4., self.L / 4., self.L / 4.)),
246                         'ConnectLayers failed to pass the KS test.')
247 
248     def test_3D_linear_shifted_z(self):
249         '''Z-test performed on total connection count'''
250 
251         self.assertTrue(self.z_test(ConnectLayers3D_tester, 'linear',
252             source_pos=(self.L / 4., self.L / 4., self.L / 4.)),
253                         'ConnectLayers failed to pass the Z-test')
254 
255     def test_3D_linear_multithread_ks(self):
256         '''KS test performed on source-target node distances'''
257 
258         self.assertTrue(self.ks_test(ConnectLayers3D_tester, 'linear',
259                                      threads=4),
260                         'ConnectLayers failed to pass the KS test.')
261 
262     def test_3D_linear_multithread_z(self):
263         '''Z-test performed on total connection count'''
264 
265         self.assertTrue(self.z_test(ConnectLayers3D_tester, 'linear',
266                                     threads=4),
267                         'ConnectLayers failed to pass the Z-test')
268 
269 
270 def suite():
271     suite = unittest.makeSuite(ConnectLayersTestCase, 'test')
272     return suite
273 
274 
275 if __name__ == '__main__':
276     runner = unittest.TextTestRunner(verbosity=2)
277     runner.run(suite())
```

References

Abbott, A. (2013). Brain-simulation and graphene projects win billion-euro competition. *Nature*.

Brette, R. and W. Gerstner (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology 94*(5), 3637–3642.

Brette, R., M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. H. Jr., M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. E. Boustani, and A. Destexhe (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *J Comput Neurosci 23*, 349–398.

Chang, G. and C. B. Roth (2001). Structure of MsbA from E. coli: A homolog of the multidrug resistance ATP binding cassette (ABC) transporters. *Science 293*(5536), 1793–1800.

Crook, S. M., J. A. Bednar, S. Berger, R. Cannon, A. P. Davison, M. Djurfeldt, J. Eppler, B. Kriener, S. Furber, B. Graham, et al. (2012). Creating, documenting and sharing network models. *Network: Computation in Neural Systems 23*(4), 131–149.

Dijkstra, E. W., E. W. Dijkstra, and E. W. Dijkstra (1970). *Notes on structured programming*. The Netherlands: Technological University Eindhoven.

Djurfeldt, M. (2012). The connection-set algebra—a novel formalism for the representation of connectivity structure in neuronal network models. *Neuroinformatics 10*(3), 287–304.

Eppler, J. M., M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig (2009). PyNEST: a convenient interface to the NEST simulator. *Frontiers in Neuroinformatics 2*(12).

Ewens, W. and G. Grant (2004). *Statistical methods in bioinformatics: an introduction*, Volume 10. New York: Springer Science+Business Media.

Gewaltig, M.-O. and M. Diesmann (2007). NEST (NEural Simulation Tool). *Scholarpedia 2*(4), 1430.

Gewaltig, M.-O., A. Morrison, and H. E. Plesser (2012). NEST by example: An introduction to the neural simulation tool NEST. In N. Le Novère (Ed.), *Computational Systems Neurobiology*, pp. 533–558. Dordrecht: Springer Science+Business Media.

Greenwood, P. and M. Nikulin (1996). *A guide to chi-squared testing*. New York: Wiley.

Hill, S. and G. Tononi (2005). Modeling sleep and wakefulness in the thalamocortical system. *Journal of Neurophysiology 93*(3), 1671–1698.

Huizinga, D. and A. Kolawa (2007). *Automated defect prevention: best practices in software management*. Hoboken, NJ: Wiley-IEEE Computer Society Press.

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks 14*(6), 1569–1572.

Kriener, B. (2012). Testing connectivity in spatial networks. Internal report.

L'Ecuyer, P. (2004). Random number generation. In J. E. Gentle, W. Haerdle, and Y. Mori (Eds.), *Handbook of Computational Statistics*, pp. 35–70. Springer-Verlag.

L'Ecuyer, P. and R. Simard (2007). TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software 33*(4), 22.

Miller, G. (2006). A scientist's nightmare: Software problem leads to five retractions. *Science 314*(5807), 1856–1857.

Nordlie, E. and H. E. Plesser (2010). Visualizing neuronal network connectivity with connectivity pattern tables. *Frontiers in Neuroinformatics 3*(39).

Panik, M. J. (2005). *Advanced statistics from an elementary point of view*. Burlington, MA: Elsevier Academic Press.

Plesser, H. E. and K. Austvoll (2009). Specification and generation of structured neuronal network models with the NEST Topology module. *BMC Neuroscience 10*, P56.

Plesser, H. E. and H. Enger (2012). *NEST Topology User Manual for NEST 2.2*.

Plesser, H. E., J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.), *Euro-Par 2007 Parallel Processing*, Volume 4641 of *Lecture Notes in Computer Science*, pp. 672–681. Springer Berlin Heidelberg.

The Human Brain Project Preparatory Study Consortium (2012). The Human Brain Project: A report to the European Commission. Lausanne. http://www.humanbrainproject.eu/files/HBP_flagship.pdf.

Wang, Y. (1993). On the number of successes in independent trials. *Statistica Sinica 3*(2), 295–312.